图灵程序设计丛书

松本行弘

编程语言的设计与实现

[日] 松本行弘 / 著
[日] 日经Linux / 编
郑明智 / 译

听Ruby之父畅谈

- 设计新语言的动机、过程中的纠结与取舍
- 隐藏在各编程语言背后的设计缘由
- Ruby开发中不为人知的故事

▶ **学习大师级程序员的思维方式**


中国工信出版集团


人民邮电出版社
POSTS & TELECOM PRESS



松本行弘

（Yukihiro "Matz" Matsumoto）

1965年生于鸟取县米子市，现居岛根县。筑波大学第三学群信息学类毕业。Ruby语言发明者，现兼任网络应用通信研究所（NaCl）研究员、Ruby协会理事长、Heroku首席架构师等职。育有三女一男，饲有一猫一狗。喜欢温泉。白羊座、O型血。

郑明智

智慧医疗工程师。主要研究方向为医疗领域的自然语言处理及其应用，密切关注大数据、机器学习和深度学习等领域。

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

TURING 图灵程序设计丛书

松本行弘编程语言的设计与实现

[日] 松本行弘 / 著
[日] 日经Linux / 编
郑明智 / 译

人 民 邮 电 出 版 社
北 京

图书在版编目（CIP）数据

松本行弘：编程语言的设计与实现 /（日）松本行弘著；郑明智译. -- 北京：人民邮电出版社，2019.8
（图灵程序设计丛书）
ISBN 978-7-115-51616-9

Ⅰ. ①松… Ⅱ. ①松… ②郑… Ⅲ. ①程序语言—程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2019)第137815号

内 容 提 要

本书由 Ruby 之父松本行弘在《日经 Linux》杂志上的连载整合而成，主要介绍了新语言 Streem 的设计与实现过程。作者从设计 Streem 这门新语言的动机开始讲起，由浅入深，详细介绍了新语言开发中的各个环节，以及语言设计上的纠结与取舍，其中也不乏对其他编程语言的调查与思考，向读者展示了创建编程语言的乐趣。

本书适合各层次程序设计人员和编程爱好者阅读。

◆ 著　[日] 松本行弘
编　[日] 日经 Linux
译　郑明智
责任编辑　杜晓静
责任印制　周昇亮

◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路11号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　印刷

◆ 开本：800×1000　1/16
印张：20.25
字数：474千字　2019年8月第1版
印数：1-3 500册　2019年8月北京第1次印刷

著作权合同登记号　图字：01-2017-3629号

定价：89.00元

读者服务热线：(010)51095183转600　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东工商广登字20170147号

译者序

本书是 Ruby 之父松本行弘最新的作品。在书中作者讲述了从头开始设计一门新的流处理语言 Streem 的过程。凭借作者在业界的影响力，Streem 语言在开发阶段就已经在 GitHub 上收获了四千多 star。

虽然 Streem 语言依然在开发阶段，不适合在正式项目中使用，但是本书最大的价值并不在于 Streem 语言本身。正如作者在文中所说，很多编程语言相关的书都是在介绍一门语言该如何使用，自制编程语言的书也仅仅涉及语言的实现，讲解语言为什么这样设计的书少之又少，而这本书就属于后者。从简单的如何为语言取名，到复杂的如何设计多线程、垃圾回收等，作者在书中事无巨细地介绍了语言设计的方方面面。对于想了解语言为什么会这样设计、语法为什么是这样的读者来说，本书是一本不可多得的宝贵资料。

练武的有武痴，而作者简直就是“语言痴”（或者“语言控”，褒义），几十年来一直孜孜不倦地研究各种语言，并且与许多语言设计者有过直接的交流，所以对很多语言都很熟悉，在书中他也旁征博引地介绍了这些语言是如何设计的。因为很多语言的语法都是相通的，所以即使不打算自己去设计新的语言，了解语言设计的知识，对语言的学习也是大有裨益的。比如 3.4 节讲解了模式匹配，对于 Java 程序员理解 Scala 语言的模式匹配功能应该很有帮助。4.4 节综合介绍了垃圾回收的几种算法，写得通俗易懂，与 Java 或者 Objective-C 等语法书一般只介绍语言中用到的垃圾回收算法相比，作者的介绍更有助于读者深入理解垃圾回收，了解各种算法的优缺点，这也许在面试时就能用得上。

作者在业界活跃几十年，学识非常渊博。第 5 章就是一个很好的体现。作者在第 5 章讲解了管道编程、CSV 文件处理、时间表示、统计以及随机数等多个主题，并围绕这些主题介绍了大量相关的知识和算法。比如 5.2 节介绍了流处理相关的 `map`、`reduce` 和 `flatmap` 等函数，这些知识对学习大数据的 mapreduce 和 spark 流处理会很有帮助；5.5 节介绍的统计算法和 5.6 节介绍的随机数生成算法让译者在翻译时也受益匪浅。这些知识说不定什么时候就能派上用场，如果你记住了 5.5 节介绍的蓄水池抽样算法，那么海量数据随机抽样之类的问题就难不倒你了。

虽然作者是业界的风云人物，但在书中毫不讳言自己不擅长哪些领域、哪些地方借鉴了他人的实现，非常平易近人，不会让人感觉高不可攀。阅读本书就像是在跟一位谆谆善诱的前辈聊天，他会告诉你自己是怎么设计语言的、为什么要这样设计、别人是怎么做的、具体要怎么实现、在自己实现不了的情况下如何借鉴别人开源的东西、如何取得许可，等等，非常有趣。作者还在书中设置了“时光机专栏”，其中添加了对正文内容经过一段时间沉淀之后的思考，这些内容也很有趣，请读者不要错过。

另外，作者在书中多次流露了“最近太忙了，这个功能有机会再实现吧”的想法，让译者也心有戚戚焉。

希望读者能在阅读本书的过程中体验到语言设计的乐趣，有所收获。

由于译者水平有限，书中恐有疏漏和错误之处，还请读者随时指正。

在此衷心感谢图灵公司的各位编辑在翻译过程中给予的帮助，感谢博学儒雅的上司王蕴韬在工作上的指导，感谢亲友杨玉生、黄旭、戎政给予的支持和鼓励，最后也要感谢一直给我家庭温暖的父母、岳父母、妹妹一家、妻子和儿子。

郑明智

2019 年 5 月于余杭南湖

前言

我曾经在 Linux 专业期刊《日经 Linux》上开设了名为“边做边学编程语言”的专栏（2014 年 4 月 ~ 2016 年 12 月），本书就是对这个专栏的系列文章加以汇总，并添加了一些新的内容编辑整理而成的。

以自制编程语言为主题的书非常多，我家的书架上也放了好几本。这些自制编程语言的书绝大部分是介绍编程语言的实现的。比如以比较简单的语言的实现作为示例，介绍如何使用 yacc 和 lex 工具制作语法分析器和词法分析器，如何实现解释器等。

不了解真实情况的人总以为编程语言的实现需要高超的技术，非常困难，但如果循序渐进地进行实际操作，就会发现其实并没有那么难。实际上，在大学的计算机科学课程中，也有不少实现语言处理器之类的作业。在以年轻人为对象的编程竞赛中，比如我长期担任评委的 U22 编程竞赛，甚至有高中生挑战自制编程语言。

抛开成见认真去做的话，你会发现编程语言的设计与实现对程序员来说是一项非常有趣的智力挑战。在这个意义上，这些关于自制编程语言的书是有其存在价值的。

但这并不是说我对这些书没有不满。这些书顶多是对编程语言的实现的解说，作为示例使用的语言也几乎都是现有语言的（过于简化的）子集。而对于创建编程语言的过程中最挑战智力也最有趣的语言设计部分，可以说完全没有涉及。

不过这也是没办法的事，可以理解。为了有效利用有限的页数，缩小主题是必然的。如果不限定为简单的语法，那么无论如何也很难讲完。再往深里说，也没有多少人有设计全新的编程语言的经验，更不用说自己设计的语言在全世界被广泛使用了，可以毫不夸张地说，根本没有这样的人。

也就是说，几乎没有人能根据自己的经验来讲述如何设计编程语言。因此，将语言的设计作为后话，优先讲解语言处理器的实现技术，可以说是必然的。

其中也有少数例外的情况，比如 C++ 设计者本贾尼・斯特劳斯特卢普（Bjarne Stroustrup）所著的 *The Design and Evolution of C++*[①] 一书。这是一本非常宝贵的书。读了这本书，你就可以了解 C++ 为何是现在的样子，它的目标是什么。不过是否能靠这本书学会语言设计的方法，那就另当别论了。

既然世界上不存在这样的书，那就只能自己去写了。幸好我拥有在全世界被广泛使用的编程语言的设计经验，有这种经验的人屈指可数。另外，我兴趣广泛，对世界上各种编程语言的设计都有所了解。再者，我在《日经 Linux》上连载过文章，还有多本书的写作经验。因此，要写关于设计编程语言的书，恐怕整个日本没有人比我更合适了吧（自吹自擂）！

有了这个想法之后，我便从《日经 Linux》2014 年 4 月刊起开设了专栏，开始发表该主题的文

① 中文版名为《C++ 语言的设计与演化》，裘宗燕译，科学出版社 2012 年出版。——译者注

章（表 1）。

从创造编程语言的动机，到思考语言设计时的内心纠葛……本书提到了很多其他同类书中不曾涉及的内容，对此我也颇为骄傲。

但是在把每个月连载在杂志上的内容汇总成书时，一个难逃的宿命就是，随着时间的推移，会出现内容前后矛盾等问题。本书也不例外。

连载时的主题如表 1 所示。其中，嵌入式 Ruby“mruby”相关的介绍（2014 年 4 月刊和 2014 年 6 月刊的一部分）以及超出本书范围的相关说明（2014 年 9 月刊 ~2014 年 11 月刊）均未被收录在本书之中。另外，我还调整了连载的前后顺序，并结合 Streem 的现状修改了部分内容。

但是，整体的文章结构与连载时相比并没有大的变动。因此，为了保持内容的连贯性以及帮助各位读者理解，我在每节的末尾添加了“时光机专栏”，这是对正文的补充。在这部分内容中，我会站在现在的角度审视过去的连载主题有哪些不足之处。

“时光机专栏”的存在，好像是向世人承认自己的能力有限一样，这令我心情复杂，但想到谁也无法预知未来，我也就释然了。

接下来就由我带领大家进入编程语言设计的世界吧！

松本行弘

2016 年 12 月

表 1　连载主题一览表

《日经 Linux》刊登期号	标题	本书
2014 年 4 月刊	第 1 期　编程语言设计入门	1-1
2014 年 5 月刊	第 2 期　语言处理器的结构	1-2
2014 年 6 月刊	第 3 期　虚拟机	1-3
2014 年 7 月刊	第 4 期　编程语言设计入门（前篇）	1-4
2014 年 8 月刊	第 5 期　编程语言设计入门（后篇）	1-5
2014 年 9 月刊	第 6 期　编程语言的类型设计 (1)	-
2014 年 10 月刊	第 7 期　编程语言的类型设计 (2)	-
2014 年 11 月刊	第 8 期　编程语言的类型设计 (3)	-
2014 年 12 月刊	第 9 期　抽象的并发编程	2-1
2015 年 1 月刊	第 10 期　21 世纪的并发语言	2-2
2015 年 2 月刊	第 11 期　设计 Streem 语言	2-3
2015 年 3 月刊	第 12 期　Streem 语言的核心	2-4
2015 年 4 月刊	第 13 期　多线程与对象	2-5
2015 年 5 月刊	第 14 期　缓存与符号	2-6
2015 年 6 月刊	第 15 期　AST	2-7
2015 年 7 月刊	第 16 期　局部变量与异常处理	2-8
2015 年 8 月刊	第 17 期　套接字编程	4-1
2015 年 9 月刊	第 18 期　CSV	5-3
2015 年 10 月刊	第 19 期　基本数据结构	4-2
2015 年 11 月刊	第 20 期　各种各样的面向对象	3-1
2015 年 12 月刊	第 21 期　Streem 的面向对象	3-2
2016 年 1 月刊	第 22 期　对象表示与 NaN Boxing	4-3
2016 年 2 月刊	第 23 期　垃圾回收	4-4
2016 年 3 月刊	第 24 期　管道编程	5-1
2016 年 4 月刊	第 25 期　管道的组成要素	5-2
2016 年 5 月刊	第 26 期　无锁算法	4-5
2016 年 6 月刊	第 27 期　时间表示	5-4
2016 年 7 月刊	第 28 期　统计基础	5-5
2016 年 8 月刊	第 29 期　再看 Streem 的语法	3-3
2016 年 9 月刊	第 30 期　模式匹配	3-4
2016 年 10 月刊	第 31 期　随机数	5-6
2016 年 11 月刊	第 32 期　数据流图	5-7
2016 年 12 月刊	最后一期　后记与拾遗	-

目录

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

第 1 章

创造一门什么样的语言

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

1-1 自己创造编程语言的意义

通过实际创造一门新的编程语言，可以学到编程语言的设计思路和实现方法。随着开源的普及，创造新编程语言的门槛一下子降低了许多。创造编程语言不仅可以提升你作为技术者的价值，而且还可以使你从中获得很大的乐趣。

大家都知道我是编程语言 Ruby 的作者，我其实还是一个编程语言迷，对编程语言的痴迷程度无人能及。Ruby 是我出于兴趣钻研编程语言的最大成果，把它称为我兴趣的副产品可能更为贴切。副产品就能如此普及看起来很了不起，但与其把它全部归功于我的实力，倒不如说运气的成分更大。Ruby 已经诞生 20 多年了，如果没有这么多年来发生的各种事情与邂逅，根本不可能有今天这样的成绩。

进入创造编程语言的世界

大家有创造编程语言的经历吗？对于有过编程经历的人来说，编程语言是非常亲切的存在，但是他们往往会认为编程语言是现成的东西，也许谁都没有想过自己去创造一门新的编程语言。这也是情理之中的事情。

与人们说话用的语言（自然语言）不同，世界上所有的编程语言都是由某个地方的某个人创造的。它们不是自然产生的，而是根据明确的意图和目的被设计并实现的。所以，如果过去没有这些创造编程语言的人（编程语言的作者），那么我们今天可能还在用汇编语言编程呢。

在人们刚开始编程时，编程语言就随之出现了，可以说编程的历史就是编程语言的历史。

本书的目标是要你自己创造一门编程语言。可能有的读者会想："现在再创造编程语言还有什么意义呢？" 我稍后回答这个问题，现在我们先来看一下编程语言的历史。

■ 个人创造编程语言的历史

早期的编程语言是由在工作中切切实实与编程语言打交道的人创造的，这些人大多就职于企业的研究所（比如 FORTRAN、PL/1 的发明）、大学（比如 LISP）以及标准委员会（比如 ALGOL、COBOL）等。也就是说，设计开发编程语言是专业人士的工作，但是这个传统随着 20 世纪 70 年代计算机的普及开始发生了变化。一些计算机爱好者在拥有了自己的计算机后，出于兴趣开始编程，甚至开始开发新的编程语言。

其中最具有代表性的就是 BASIC 语言。BASIC 语言原本是美国达特茅斯学院用于教学的编程

语言，它的语法非常简单，用极少的代码实现了最基本的功能，所以深受 20 世纪 70 年代编程爱好者的喜爱，并被他们广泛使用。

这些编程爱好者也开始开发自己版本的 BASIC 语言。当时，个人计算机[①]的内存顶多几千兆，他们开发的 BASIC 语言就是可以在内存如此之小的机器上工作的小规模版本。这些小规模的 BASIC 程序大小不到 1 KB，它们在 4 KB 左右的内存上也能工作，跟现在需要大内存的语言处理器比起来真是令人惊讶。

微机杂志的时代

以个人开发的 BASIC 为代表的小规模语言（Tiny 语言）处理器不久便以各种各样的形式进行了发布。当时的软件有的以 Dump list 的形式刊登在计算机杂志上，有的将程序数据进行音频转换后收录在杂志附带的薄膜唱片（sonosheet）中发布。现在的人恐怕已经不知道薄膜唱片了吧。薄膜唱片是指塑料做的薄薄的唱片，不过唱片这个词几乎没有人用了。据说当时的计算机爱好者都用唱片播放器连接计算机来读取数据，而不使用磁带录音机这个最普遍的外部存储设备。

20 世纪七八十年代是计算机杂志（当时称为微机杂志）的全盛时期，在日本以下 4 种杂志竞争激烈。

- *RAM*（广济堂出版）
- *My Computer*（电波新闻社）
- *I/O*（工学社）
- *ASCII*（ASCII 公司）

这 4 种杂志中现在只有 *I/O* 仍在发行，不过也大不如前了。作为一个了解当时情况的人，我的内心充满了无限感慨。

这之后，*My Computer* 杂志派生出了 *My Computer BASIC Magazine*，又发生了很多事情，继续讲下去恐怕就会变成上岁数人的叙旧了，所以点到为止吧。如果去问问现在三四十岁的程序员，相信他们中间很多人都会眉飞色舞地讲起那个年代的事情。

当时的微机杂志附带了收录 BASIC 的薄膜唱片，除此之外还介绍了其他几个小规模语言，如 GAME、TL/1 等。这些语言都反映了当时那个时代的特色，非常有趣，我会在本节的最后对其进行介绍，请大家务必读一读。

个人创造编程语言的现状

为什么从 20 世纪 70 年代后期到 80 年代前期开始兴起个人创造编程语言了呢？我认为最大的

① 通常称为微机。微机是微型计算机、微型机的简称。

原因是当时难以获取开发环境。

20 世纪 70 年代后期广泛使用的微机是 TK-80（图 1-1）那样的主板裸露在外的单板机，很多都是半成品，需要自己去钎焊。这样的机器不可能自带开发环境之类的东西，软件都要自己输入机器语言之后才会工作。

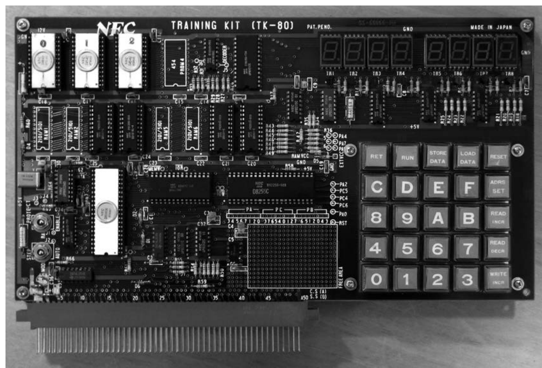


图 1-1　TK-80

20 世纪 70 年代末期才出现 PC-8001 和 MZ-80 那样的“成品计算机”。然而，这种计算机顶多带一个 BASIC 开发环境，因此人们很难自由地选择开发语言。虽说市面上也有商用的语言处理器，但 C 编译器的定价就要 19.8 万日元，这不是普通人可以轻易买得起的。于是，人们便有了热情去创造一门自己的编程语言。

可现在获取语言的开发环境已经不再是麻烦事了。各种编程语言和开发环境作为开源软件被公开，即使是非开源的，也可以轻松地通过网络得到免费版本。

这样一来，现在自己创造编程语言岂不是没有任何意义吗？如果这个问题的答案为“是”，那么本书在第 1 章开头就结束了。

我认为（而且为了这本书也应当这么回答），这个问题的答案为“否”。即使是现在，自己创造一门新的编程语言也是有意义的，而且有很重要的意义。

而且现在很多广泛使用的编程语言也都是在开发环境容易获取的情况下，由个人设计和开发出来的。如果个人开发编程语言真的没有意义，那么 Ruby、Perl、Python 和 Clojure 这些语言也就不会诞生了。

不过即便如此，我认为 Java、JavaScript、Erlang 和 Haskell 这些语言也可能会以其他形式出现，因为它们会作为业务和研究的一环被开发出来。

■ 为什么要创造新的编程语言

那么如今个人设计开发编程语言的动力究竟是什么呢？

回顾我自身的经历以及参考其他语言作者的意见，我认为有以下几点理由。

- 提高编程能力
- 提高设计能力
- 打造个人品牌
- 获得自由

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

首先，编程语言的实现可以说是计算机科学的综合艺术。作为语言处理器的基础，词法分析和语法分析也可以应用在网络通信的数据协议的实现等方面。

实现语言功能的库和实现其中的数据结构，这正是计算机科学要做的事情。尤其是编程语言的应用范围广泛，很难事先预测会被用于什么方面，因此库和数据结构的实现难度也就更大，但也变得更加有意思了。

另外，编程语言还是人与计算机间的接口。设计这样的接口，就需要深入考察人是如何思考问题的、下意识中有什么样的期待。反复进行这样的考察，对编程语言之外的应用程序接口（API）设计、用户界面（UI）设计，甚至用户体验（UX）设计都是有益的。

提升个人品牌

也许有人会感到意外，实际上在 IT 行业，对编程语言感兴趣的人不在少数。这是毋庸置疑的，因为编程与编程语言有着切不断的关系。以编程语言为主题的活动和会议等往往都会吸引很多人参加，由此我们也能感受到编程语言的魅力。正因如此，很多人在网上发现新的语言后就会开始尝试。就拿 Ruby 来说，它在 1995 年被发布到网上之后，仅仅 2 周左右就吸引了 200 多人加入邮件列表，着实令人惊讶。

可是，虽然有很多人愿意尝试使用新的编程语言，却几乎没有人会去设计并实现一门编程语言，而且是超越杂志提及的“小儿科语言”那种程度的能够实用化的编程语言。但我保证，仅凭设计出一个实用的编程语言这一点，你就会得到人们的尊敬。

在这个开源的时代，技术人要想生存下去，在技术社区的存在感是非常重要的。虽然技术人只要开源其软件就能达到站稳脚跟的效果，但编程语言的“特殊感”会进一步提升其品牌效应。

乐趣第一

另外，编程语言的设计与实现比任何事情都更有趣。的确如此。与计算机科学相关的具有挑战性的工程也是这样。设计编程语言还可以帮助使用这门语言的程序员思考，甚至左右他们的想法，这一点也非常有意思。

通常来说，编程语言有一种从别处获取的、不容侵犯的感觉。如果是自己创造编程语言，就完全没有这个问题。你可以按照自己的喜好进行设计，如果不满意或者有更好的想法，也可以自由地修改。从某种意义上来说，这是终极的自由。

编程在某种意义上是对自由的追求。通过亲自编程，我们可以获得单纯使用他人的软件时享受不到的自由。至少对我来说，这是编程的一个重要动机。于我而言，创造编程语言是获取更高程度自由的手段，也是我的乐趣与快乐的源泉。

为什么创造新编程语言的人不多

虽说自己创造一门编程语言有这么多好处，但并不是每个人都会去做。正如上文所说的那样，对编程语言感兴趣的人虽然有一些，但着手去创造编程语言的人几乎没有。说是“感兴趣的人有一些”，但从占总人口的比例来看，其实少到可以算作误差范围的程度，更不用说有动力去创造新编程语言的人了，就算没有也不足为奇。

我自己在关注编程语言几年后就着了迷，但是在进入大学主修计算机科学之后，才注意到并不是所有人都对编程语言感兴趣。这是因为我在偏僻的乡下长大，周围没有喜欢编程的人可供比较。这一点对我来说也不知道是幸还是不幸。

“难道我跟别人不一样？”意识到这一点的时候，我很震惊。因为当时的微机杂志上刊登了很多关于 TL/1 等编程语言的文章。我本以为对编程感兴趣的人（和我一样）很可能也会对编程语言着迷，但实际上并非如此。

本来就对编程语言不感兴趣的人自不用说，即使是感兴趣的人，也很难走到自己设计并实现编程语言这一步。

关于这个问题的原因，我思考过很长时间。作为编程语言设计者，在参加编程语言相关的活动时，我也曾以过来人的身份鼓励别人尝试一下，但结果总是不尽如人意。当然，万事开头难，开始一件新的事情是需要很大勇气的。但即使是这样，反响也太差了。

没必要想得很难

问了很多人之后，我才知道大家为什么不去着手尝试了。那是因为就算有兴趣创造一门新的编程语言，在开始之前多半也会有某种心理障碍，也就是觉得“编程语言有现成的，本来就不需要自己去设计和开发”。难得有那么几个人不会产生这种心理障碍，却又觉得语言的实现似乎很难。也就是说，他们觉得编程语言很有趣，自己也想做做看，却不知道如何去实现。

仔细想来，关于编程语言的实现的书虽然出乎意料地出版了很多，但大部分都是大学教材的难度，非常不容易理解。另外，与编译原理有关的“文法类型”和“Follow 集合”等晦涩的术语也频繁出现。

但是认真想一想，我们的目的是出于兴趣创造自己的编程语言，而不是去掌握编程语言的实现所需的所有知识。如果你认为在没有完全掌握正确的知识之前就无法着手创造编程语言，那就大错特错了，你的热情会被逐渐消磨殆尽。

成就一番伟大的事业首先需要的就是热情，不能保持热情是不行的。一旦有了创造编程语言的热情，就应尽快开始，以后再根据需要慢慢地掌握所需的知识即可。

本书主要介绍创造简单的语言处理器所需要的基本知识以及工具的使用方法，并不涉及编程语言实现的较难部分。相较于理论背景，我更想把重点放在如何设计编程语言上。

微机杂志中介绍的Tiny语言

GAME

GAME(General Algorithmic Micro Expressions)是由BASIC派生的Tiny语言。它最大的特征是关键字全部是符号，以及所有的语句都是赋值语句。

例如赋值给“?”时会输出数值，反过来将“?”赋值给变量时会要求输入数值。字符串的输入输出使用“$”。另外，将行号赋给“#”时为goto语句，将行号赋给“!”时为gosub语句(调用子程序)。

另外，像"ABC"这样的一个字符串语句会打印出字符串，后面有“/”的话会换行。

这是一门非常有意思的编程语言，示例代码如图1-A所示。它既像BASIC，又不像BASIC，请大家好好感受一下。

GAME是一门非常简洁的语言，用8080汇编语言编写的解释器的大小还不到1 KB。另外，由中岛聡(当时居然还是高中生)开发的使用GAME编写的GAME编译器，代码仅有200行左右。真不知道我们应该惊叹GAME的语言表现能力，还是中岛聡的技术能力。

TL/1

同一时期在*ASCII*杂志上发表的Tiny语言中还有TL/1(Tiny Language/1)，它的名字应该是模仿了美国IBM公司开发的编程语言PL/1。与受BASIC影响使用符号的GAME语言不同，TL/1拥有类似于Pascal的语法，让人觉得更加“正常”。另外，TL/1的语言处理器是编译型的，与主体为解释型的GAME比起来速度更快。但实际上GAME也有编译器，这一点我们在前文中介绍过。

TL/1的特征是语法类似于Pascal，以及变量类型只有1字节的整数。各位读者也许会想这样怎么编写代码，不过当时的主流CPU是8位的，所以TL/1设计成这样也不是很怪异。虽说是运行在8位CPU上，但包括GAME在内的其他语言都提供了16位的整数类型。

那么1字节无法表示的超过255的数值该如何编写呢？答案是按字节进行分割，用多个变量组合表示。比如用2个变量保存16位整数，边看计算溢出时的进位标志边计算(图1-Ba)。在当时的8位CPU上，大部分处理用16位整数就已经足够了(地址用16位的话就可以访问所有地址空间)。作为Tiny语言，这样的功能已经足够了。各位读者如果有兴趣，可以显式地查看进位标志，使用多个变量进行24位计算或32位计算。

可以处理指针和字符串

另外，指针也无法仅用1字节来表示。这里用mem数组进行访问，也就是说，用下面表

达式中hi,lo表示的16位地址来访问指定地址的内容。

```
mem(hi, lo)
```

下面是将地址的值替换为v。

```
mem(hi, lo) = v
```

当时的个人计算机（微机）最多只有32 KB的内存，所以能用16位地址访问就已经足够了。

还有就是字符串。当然，我们也可以将字符串当作字节数组，对每个字节依次进行操作，但是这样处理太麻烦，因此TL/1设计了用于数据输出的WRITE语句。

例如，用TL/1开发的Hello World程序如图1-Bb所示。TL/1中变量本应只有1字节的整数，却出现了字符串。实际上，WRITE是为了能够处理字符串而单独增加的语法。

WRITE之外的语句是无法处理字符串的，所以不能进行普通的字符串处理，只能够操作1字节的整数。现在看来可能会觉得很不可思议，但是在不属于Tiny语言的Pascal和FORTRAN中，输入输出也是被特殊处理的，这在当时也许是一种比较普遍的做法。

```
100---------------- Comment -------------------
110|     如果紧接在行号之后的不是空白，则将该行作为注释
120--------------------------------------------
130
200 / " FOR循环语句 是 变量名=初始值，最终值  ... @=(变量名 + 步长) " /
210  A=1,10
220    ?(6)=A
230  @=(A+1)
240
300 / " IF语句的例子 " /
310  B=1,2
320    ;=B=1 " B=1 " /
330    ;=B=2 " B=2 " /
340  @=(B+1)
350
400 / " 数值输入与计算 " /
410  "A = ?" A=?
410  "B = ?" B=?
420  "A + B = " ?=A " + " ?=B " = " ?=A+B /
```

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

```
430  "A * B = " ?=A " * " ?=B " = " ?=A*B /
440
500 / " 数组与字符输出 " /
505--------令数组的地址为$1000
510  D=$1000
520  C=0,69
525--------作为2字节数组写入
530    D(C)=(C+$20)*256+C+$20
540  @=(C+1)
560  C=0,139
570--------作为1字节数组读取，并输出为字符
580    $=D:C)
590  @=(C+1)
600
700 / " GOTO 与 GOSUB " /
710  I=1
720  I=I+1
730  !=1000
731*  ?(8)=I*I
740  ;=I=10 #=760
750  #=720
760
900 / "程序结束 " /
910 #=-1
920
1000 / " 子程序 " /
1010 ?(8)=I*I
1020 ]
```

图 1-A　GAME 语言的示例代码

```
% (a)
% 以"%"开始的行是注释，当时不能使用日语
BEGIN
  A := 255
  B := A + 2    % overflow
  C := 0 ADC 0  % add with carry
END

% (b)
```

```
BEGIN
  WRITE(0: "hello, world", CRLF)
END
```

图 1-B　TL/1 语言的示例代码

时光机专栏

我原本打算改造mruby

本节相当于从 2014 年 4 月刊开始连载的第一期内容。我热忱地讲述了编程语言的设计。

在第一期的时候，我还没有想好要设计开发一门什么样的编程语言。当时我打算改造一下自己编写的语言处理器 mruby，因此在连载时介绍了 mruby 源代码的获取方法以及代码目录结构等内容。不过在实际操作中完全没用上 mruby 的源代码，所以本书省略了这部分内容。

虽然本书不会涉及这部分内容，但由于 mruby 比较简单，所以它还是很适合作为编程语言的实现的教材来使用的。如果有读者想阅读 mruby 的源代码进行学习，可以从 http://www.mruby.org/ 这个网址开始各种尝试。另外，mruby 的源代码可以从 GitHub 网站（https://github.com/mruby/mruby）上下载。

如果在学习的过程中有什么疑问、意见，或者发现了错误，请通过 GitHub 的问题追踪器报告给我们。现在 mruby 的开发正朝着国际化方向发展，因此建议使用英语描述问题。另外，大家也可以通过我的推特账号（@yukihiro_matz）用英语或日语与我交流。

1-2 语言处理器的结构

本节将简单地讲解编程语言与语言处理器的关系以及语言处理器的结构，为开始设计编程语言做准备。我们首先会制作一个计算器程序，同时也将以mruby的实现为例，介绍一下实用的语言处理器。

虽说我们要创造一门编程语言，但具体要做什么，恐怕没有多少人能回答得上来吧。这是因为大部分人只去学习现有的语言，从未考虑过设计一门语言。

语言和语言处理器

编程语言拥有多层构造。首先，在大的层面上，可将编程语言分为表示交流规则的“语言”和处理此语言使其在计算机上运行的“语言处理器”。很多人在使用“编程语言”这个词时，往往都会将语言和语言处理器混同起来。

语言是由语法和词汇构成的。语法是一种规则，规定了在该语言中如何表述才能使程序有效；而词汇是能从使用该语言编写的程序中调用的功能的集合，之后会以库的形式逐渐增加。在设计语言的场景中说起词汇，就是指该语言一开始就具备的内置功能。

不知道大家有没有注意到，在定义语法和词汇的过程中并没有用到软件。构思设计“我心中的最强语言”是不需要使用计算机的。实际上，我在乡下读高中时，还没怎么掌握编程技术，但想着将来有一天或许自己要设计开发编程语言，就用自己瞎想的编程语言在记事本上写了很多程序。以前回老家时也找过那时候的记事本，但是怎么也找不到，估计是扔掉了吧，想想就觉得可惜。我也不记得是什么样的语言了，不过好像是受了 Pascal 和 Lisp 的强烈影响写出来的。

语言处理器是能够使语法和词汇在计算机上实际运行的软件。要想使编程语言成为真正的语言，而非仅仅停留在一个想法上，是离不开语言处理器的。无法运行的编程语言在严格意义上不能称为编程语言。

语言处理器的结构

当你打算制作语言处理器时，如果不了解语言和语言处理器究竟是什么结构，就无法实现你的创作愿望。方便起见，这里我们使用现成的语言处理器来介绍一下语言处理器的结构。我们先不拘泥于技术细节，来了解一下语言处理器的概况。

语言处理器是计算机科学的集合，是一款非常有意思的软件。计算机专业的大学生应该多少都学过语言处理器的制作方法，可以说语言处理器是计算机科学的基础（之一）。这就能够解释为什么有关语言处理器的图书比比皆是了。

但是很多介绍自制编程语言的方法的书，都将过多的笔墨用在了介绍语言处理器的制作方法上，几乎没有一本书涉及语言设计的相关知识。可能这些书所说的“自制编程语言的方法”就等同于“语言处理器的制作方法”，因此这么写也无可厚非。这些书的目的是教给你自制编程语言的方法（即语言处理器的制作方法），至于你是否真的会去制作，则不是它们要考虑的范围。

而本书将焦点放在了语言的设计上。不过，像曾经的我那样只是在记事本上写下自己空想的“理想语言”是没有现实意义的，因此我会先讲解一下语言处理器的基础知识作为导入。

首先我们来了解一下语言处理器的构成。

语言处理器的构成

语言处理器大体上可分为解释语法的“编译器”、相当于词汇的“库”，以及实际运行软件所需的“运行时（系统）”。这三大构成要素的比重会因语言和处理器性质的不同而发生变化（图 1-2）。

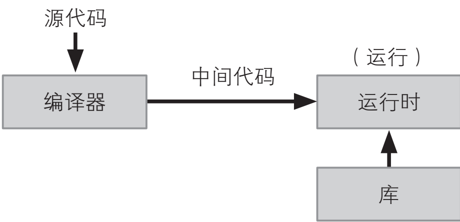


图 1-2　语言处理器的构成要素

早期出现的语言，比如 TinyBASIC 这样简单的语言，语法较少，编译器基本不做什么事情，主要的处理都在运行时完成。这样的处理器称为“解释器”（interpreter）（图 1-3）。

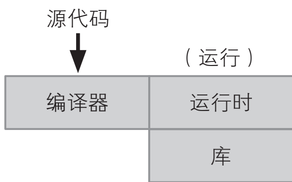


图 1-3　BASIC 语言处理器

编译器与运行时一体化的“解释型”的例子。在很多情况下，库也不单独分开

但是这样纯粹的解释型语言越来越少了。现在很多语言的处理器都是先将程序编译为内部代码，再在运行时执行内部代码。当然 Ruby 也是其中之一。这种“编译器 + 运行时”的组合形式，看起来像源代码未经转换就被直接执行了，因此有时也被称为“解释型”。

另外，像 C 语言这种在与机器非常接近的层面上追求效率的语言，乎不存在运行时，只有解释语法的编译器部分非常突出，这样的语言处理器被称为“编译型”（图 1-4）。语言处理器的构成要素与语言处理器自身的分类同名，这容易让我们感到混乱。在 C 语言这类语言中，作为转换结果的程序（可执行文件）是可以直接运行的软件，所以不需要负责运行的运行时。部分运行时的工作，比如内存管理等，由库和操作系统的系统调用负责。

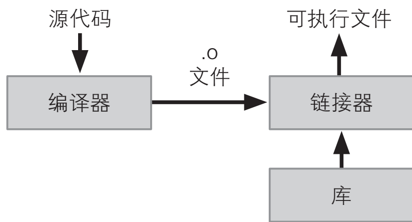


图 1-4　C 语言处理器

输出可执行文件的“编译型”的例子。因为可执行文件可以被直接运行，所以几乎没有等同于运行时的部分，库负责了一部分运行时的工作（内存管理等）

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

在语言处理器中，既有像Ruby这样“表面看上去是解释型但内部有编译器在工作”的语言处理器，也有像Java这样“表面看上去是编译型但内部有解释器（虚拟机）在工作”的语言处理器。Java是一种“混合型”的语言，它将程序转换为虚拟机的机器码（JVM字节码），并由虚拟机（JVM）来执行（图1-5）。

另外，Java为了提高运行效率，运行时采用了将字节码转换为机器码的即时编译（Just In Time Compiler）等技术，变得越来越复杂。

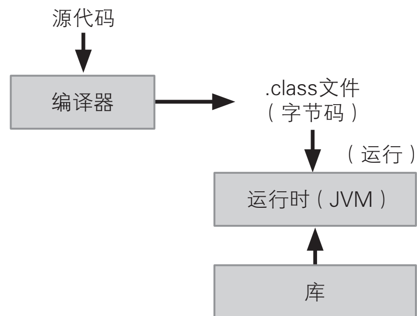


图1-5 Java语言处理器

虚拟机编译器的例子。编译器输出虚拟机的机器码（字节码），由运行时（虚拟机）负责执行

编译器的构成

接下来，我们看一下语言处理器的各个构成要素的内部构造。首先是编译器。

编译器的工作是将编程语言的源代码转换为可执行的形式。

很多编译器都会把转换处理分成多个阶段进行，按照源代码由近及远的顺序分为“词法分析”“语法分析”“代码生成”“优化”。不过，这只是一个大致的分类，并非所有编译器都会运行所有阶段。

(1) 词法分析

词法分析简单来说就是“将源代码由字符序列转换为有意义的单词（token）序列”的工序。将只是字符串的源代码整理为有些许意义的单词序列，后续阶段的处理就会变得简单。比如，将Ruby程序

```
puts "Hello\n"
```

进行词法分析，转换为

```
标识符（puts）字符串（"Hello\n"）
```

我会在后面介绍语法分析时解释单词序列的意思。

词法分析的处理通常按照如下顺序进行：语法分析器调用函数请求下一个单词时，词法分析器从函数内部的源代码中将字符逐个取出，整理为一个单词后，返回下一个单词。

我们可以借助lex工具根据编写单词的规则自动生成词法分析函数。例如，为数字和简单的四则运算生成词法分析函数的lex程序如图1-6所示。

看一下数值处的规则就会明白，在编写构成单词的模式时可以使用正则表达式。这个例子中

虽然只有运算符、数值和空格等，但在这条延长线上还可以增加各种各样的单词。这个 lex 程序（假定保存为 calc.l 文件）用 lex 执行后会生成名为 lex.yy.c 的 C 文件。编译这个文件就可以使用 yylex() 函数进行词法分析了。

```
%%
"+"              return ADD;
"-"              return SUB;
"*"              return MUL;
"/"              return DIV;
"\n"             return NL;

([1-9][0-9]*)|0|([0-9]+\.[0-9]*) {
    double temp;
    sscanf(yytext, "%lf", &temp);
    yylval.double_value = temp;
    return NUM;
};

[ \t] ;

. {
    fprintf(stderr, "lexical error.\n");
    exit(1);
}
%%
```

图 1-6　为计算器编写的 lex 程序 calc.l

像这样，使用 lex 即可简单地实现词法分析，但 mruby 并没有用 lex。这是因为 Ruby 中根据语法分析确定的状态，即使是字面相同的文字，有时也会产生不同的单词。实际上 lex 也能写出带状态的词法分析函数，不过自己编写也不是什么特别难的事情，我也想尝试去写写看，所以就没有使用 lex。写 Ruby 的时候，我还很年轻。

(2) 语法分析

语法分析是检查在词法分析阶段准备好的单词是否符合语法，并进行符合语法的处理的工序。

语法分析的方法有好几种，其中最有名、最简单的方法是使用别名为“生成编译器的编译器”的语法分析函数生成工具，比如 yacc（yet another compiler compiler）。mruby 也用了 yacc，准确来说是用了 yacc 的 GNU 版本 bison。除了 yacc 之外，生成编译器的编译器还有 ANTLR 和 bnfc 等，这里就不一一介绍了。

yacc 中，编译器解释的语法是根据 yacc 编写规则编写的。例如，计算器输入的语法如图 1-7 所示。

从开头到 %% 的部分是定义部分，定义单词的种类和类型。另外，%{ 和 }% 之间的部分由于直接插入到了所生成的 C 语言程序中，所以会进行头文件的引用等。

%% 与 %% 之间的部分是计算器的语法定义。这个定义是以语法定义规范巴科斯范式（Backus-Naur Form，BNF）的写法为基础的。%% 之后的部分也被直接插入到 C 语言程序中，因此动作（action）部分调用的函数的定义等会被放置在这里。

查看计算器的语法

接下来我们看一下计算器的语法。第一个例子我会选用非常简单的语法来介绍，与普通的计算器一样，这里没有运算符的优先顺序。

我们从第一个规则开始看起。BNF 是规则的描述，默认第一个规则在前面。第一个规则如下所示。

```
program : statement
| program statement
;
```

从这个规则中我们可以看出，正确的计算器语法为 program，意为“program 的定义是在 statement 或 program 之后紧跟着 statement”。“:”是定义，“|”是“或者”，而单词的排列意味着各部分的定义会连接起来。在这个规则中，单词 program 也出现在了右侧，形成了递归，这没有关系。yacc 中就是像这样利用递归来写循环的。

接着来看下一个规则。

```
statement : expr NL
;
```

这个规则的意思是“statement 的定义是在 expr 之后紧接着 NL”。这里没有定义 NL 的意思，它是词法分析器碰到回车时传过来的单词。

接着再看 expr 的定义。

```
expr : NUM
| expr ADD NUM
| expr SUB NUM
| expr MUL NUM
| expr DIV NUM
;
```

```
%{
#include <stdio.h>

static void
yyerror(const char *s)
{
  fputs(s, stderr);
  fputs("\n", stderr);
}

static int
yywrap(void)
{
  return 1;
}

%}

%union {
    double double_value;
}

%type <double_value> expr
%token <double_value> NUM
%token ADD SUB MUL DIV NL

%%

program    : statement
           | program statement
           ;

statement  : expr NL
           ;

expr       : NUM
           | expr ADD NUM
           | expr SUB NUM
           | expr MUL NUM
           | expr DIV NUM
           ;
%%

#include "lex.yy.c"

int
main()
{
  yyparse();
}
```

图 1-7　计算器语法分析 calc.y

这个规则的意思是“expr 的定义是 NUM（表示数值的单词），或者在 expr 之后紧跟着运算符，再之后跟着数值”。这里也用递归实现了循环。因此，“1”是数值，所以是 expr。“1+1”是 expr“1”与运算符“+”以及数值的排列，所以也是 expr。同理，“1+2+3”等也都是 expr。大家可以大概想象出 BNF 的结构了吗？

我们试着运行一下前面编写的计算器程序（图 1-8）。用 lex 执行图 1-6 的程序，用 yacc 执行图 1-7 的程序之后，对生成的名为 y.tab.c 的 C 源文件进行编译，就完成了计算器的语法检查。计算的部分完全没有实现，所以只进行了语法检查。如果输入的代码语法正确就什么也不做，如果语法错误就会显示 syntax error 并结束运行。

```
% lex calc.l ⏎          ◀—生成词法分析代码
% yacc calc.y ⏎         ◀—生成语法分析代码
% cc y.tab.c ⏎          ◀—编译
% a.out ⏎               ◀—运行
1 + 1 ⏎           ◀—输入正确的表达式
2 ⏎       ◀—这个也是正确的表达式
1 + ⏎     ◀—尝试错误语法
syntax error      ◀—显示错误并结束
%
```

⏎这个是回车标记

图 1-8　计算器程序的编译和运行

实现计算器程序

不能进行计算的计算器是没有意义的，所以我们让它来实际计算一下。在 yacc 中，我们可以编写规则成立时运行的动作。也就是说，在图 1-7 的 yacc 代码中添加实际的动作进行计算和显示，计算器就完成了。具体来说，就是将图 1-7 程序中的 statement 和 expr 的规则部分替换为图 1-9 的代码。

在这个计算器的例子中，计算和显示直接在动作部分进行，也就是所谓的“纯粹的解释器”。然而在实际的编译器中很少这样直接进行处理，因为无法支持循环以及用户自定义的函数。例如 mruby 中创建了表示语法结构的树结构，并传递给后续的代码生成处理。我们来看一个创建树结构的例子：将 mruby

```
statement  : expr NL
               {
                 fprintf(stdout, "%g\n", $1);
               }
           ;

expr       : NUM
           | expr ADD NUM
               {
                 $$ = $1 + $3;
               }
           | expr SUB NUM
               {
                 $$ = $1 - $3;
               }
           | expr MUL NUM
               {
                 $$ = $1 * $3;
               }
           | expr DIV NUM
               {
                 $$ = $1 / $3;
               }
           ;
```

图 1-9　计算器程序的动作

的 `if` 语句转换为图 1-10 中的 S 表达式（本质是结构体的链接）。

```
# 将如下的Ruby程序
if cond
  puts "true"
else
  puts "false"
end

# 转换为如下的S表达式
(if (lvar cond)
    (fcall "puts" "trued")
    (fcall "puts" "false"))
```

图 1-10　mruby 语法树的结构

(3) 代码生成

在代码生成处理中，除了遍历语法分析处理中生成的树结构之外，还会生成虚拟机的机器码。

好像自 Java 虚拟机兴起后，这种“虚拟机的机器码”就多被称为“字节码”。的确，Java 虚拟机的机器码是以字节为单位的，所以叫字节码也无可厚非（其前身 Smalltalk 也是以字节为单位的字节码）。但 mruby 的机器码是 32 位的，因此称它为“字码”（word code）可能更合适。由于字节码这个称谓不够准确，字码这一术语又不常用，所以在 mruby 内部就称为 iseq（instruction sequence，指令序列）。另外，在 iseq 上附加了符号（symbol）等信息的程序信息（代码生成的最终结果）被称为 irep（internel representation，内部代码）。

mruby 的代码生成处理并没有做很难的分析。如果想深度分析，在语法分析处理的动作部分直接生成代码也是可能的。但 mruby 出于各种原因还是将代码生成处理拆分为了几个阶段，采用了类似于 S 表达式的树结构作为中间代码。

mruby 中对代码生成处理进行了拆分

这么做的第一个原因是考虑到了可维护性。在语法分析的动作部分的确可以生成代码，程序的大小也会因此而缩小（尽管缩小得不多），但语法分析与代码生成的一体化会使程序变得更加复杂，问题也不易被发现。

动作部分是根据与规则的模式匹配的顺序被调用的，因此相比程序化的动作，运行顺序难以预测，调试起来也非常困难。考虑到可维护性，采用在动作部分只生成语法树这种简单的结构才是明智之举。

这样一来，对于嵌入式 mruby 来说，内存使用量的增加将令人担忧，但幸运的是编译部分（包含语法分析和代码生成部分）可以在运行的时候被分离出来。也就是说，将 Ruby 程序预先转换为 irep，那么在运行的时候就不需要进行编译了。这样处理有助于节约内存，即使在内存容量较小的环境中，我们也不必过于担心内存使用量。

将语法分析结果的树结构进行代码生成处理后，就会生成图 1-11 那样的 irep。ireq 原本是二进制文件（结构体），不易理解，因此就把它转换成这种我们能理解的形式了。

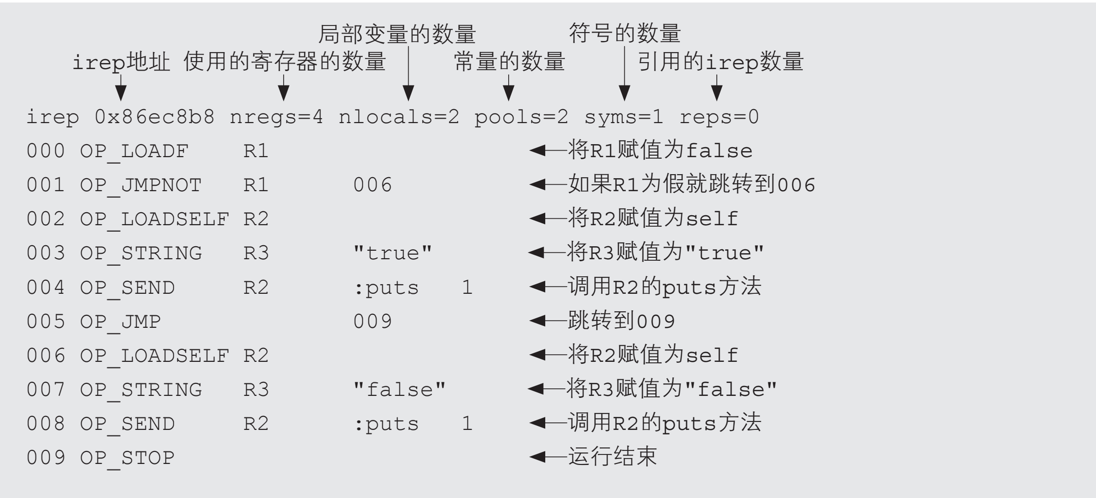


图 1-11　代码生成结果（irep）

(4) 优化

根据实现方式的不同，编译器有时会在代码生成前后进行优化处理。在 mruby 的情况下，由于 Ruby 语言的特性使其很难进行优化，所以只在代码生成处理的过程中进行极少的优化。

这种优化被称为“窥孔优化”（peephole optimization），在指令生成时仅参考正前方的指令来进行可能的优化。mruby 编译器实施的部分优化如表 1-1 所示。

表 1-1　mruby 的优化

类别	原来的指令	优化后
删除没有意义的赋值	R1=R1	删除
删除交换指令	R2=R1;R1=R2	R2=R1
削减赋值	R2=R1; R3=R2	R3=R1
削减赋值	R2=1; R3=R2	R3=1
删除重复的 return 指令	return; return	return

编译处理之后

mruby 在编译处理结束之后执行的处理有两种：一种是直接运行编译结果，运行时使用 mruby 适用的虚拟 CPU，在运行虚拟 CPU 时，也会使用对象管理等运行时和库；另外一种是将编译结果写到外部文件，这样就能够生成直接链接编译结果的程序，能够在去掉编译器的状态下执行 Ruby 程序，这对于内存限制严格的嵌入式系统来说是一个很有效的方法。

小结

本节介绍了语言处理器的构成，但并没有涉及语言设计的内容，虽然我对此感到不太满意，但为了能介绍得更详细，也只好这样了。

时光机专栏

讲解语言处理器是一件困难的事情

本节是杂志 2014 年 5 月刊中刊登的内容，介绍了语言处理器相关图书中都会提到的 yacc 的使用方法等。其中用了很老的计算器程序作为示例，令我有些汗颜。

不过，值得肯定的一点是，除了计算器这种“小儿科”的程序之外，本节还介绍了 mruby 这个实用的语言处理器的构成。之所以介绍这部分内容，是因为无法用计算器程序讲解代码生成和优化。

虽说如此，本节也仅限于向大家介绍了“有这种东西存在”，对此我有些遗憾。让我感到左右为难的是，过于详细讲解 mruby 的实现会使内容变得太难，可是不提的话自己又觉得不满意。真是难以抉择。

本节介绍的 yacc 编写规则会在后面介绍 Streem 的实现时多次出现，届时本节的内容就会起到作用。

1-3 虚拟机

本节将介绍编程语言处理器的核心部分——虚拟机（Virtual Machine，VM）的实现。在介绍完用于实现虚拟机的四大技术之后，我们将看一下mruby的虚拟机实际拥有的指令。

我们在 1-2 节中讲过，运行源代码编译结果的是运行时。运行时有多种实现方法，本节要讲的虚拟机就是其中之一。

用软件实现的 CPU 来运行

虚拟机这个单词有多种不同的含义，本节中指“用软件实现的（无实际硬件的）计算机”。

这与在虚拟机软件和云计算等语境中出现的虚拟机的含义不同。在虚拟机软件等语境中，虚拟机是指通过把实际存在的硬件用某种软件封装进行虚拟化，从而实现多个系统的同时运行以及系统在硬件间的迁移。维基百科中把这种虚拟机归类到了“系统虚拟机”中，而把本节所要介绍的虚拟机归类到了“进程虚拟机”中。

Ruby 到版本 1.8 为止都没有实现（进程）虚拟机，而是通过遍历编译器生成的语法树（支持用指针链接起来的结构体所实现的 Ruby 程序语法的树结构）来运行程序的（图 1-12）。这种方法虽然非常简单，但每执行一个指令都要访问指针，成本不容小觑。在 Ruby 1.8 出来之前大家都说 Ruby 很慢，这就是其中一个原因。

```
int
vm(node* node) {
  while(node) {
    switch (node->type) {
    case NODE_ASSIGN:
      /* 赋值处理 */
      ...
      break;
    case NODE_CALL:
      /* 方法调用处理 */
      ...
      break;
    ...
    }
    /* 跳到下一个节点 */
    node = node->next;    /* ← 这里慢 */
  }
}
```

图 1-12　语法树解释器（概要）

为什么以前的 Ruby 很慢

我觉得需要说明一下为什么这么简单的结构运行速度会那么慢。大家都知道硬盘的访问速度要比内存的访问速度慢很多，可内存的访问速度又如何呢？大家平常写代码时，很少会注意内存的速度吧。

但实际上，CPU 与内存之间的距离出乎意料地远。与 CPU 的执行速度相比，通过内存总线读取指定地址的数据的速度要慢很多。在访问内存时，CPU 只能等待数据的到来，这个等待时间就会对执行速度产生影响。

为了削减这样的等待时间，CPU 中内置了“内存缓存”（memory cache）的机制，该机制简称为“缓存”。缓存是 CPU 电路中嵌入的小容量的高速内存。通过事先将数据从主存读取到缓存中，把对内存的读写转化为对高速缓存的读写，能够削减访问内存的等待时间，提高处理速度。

由于缓存必须嵌入到 CPU 内部，所以其容量有着严格的限制，能够预先读入的数据很少[①]。为了有效利用缓存，需要把接下来要访问的内存空间事先读取到缓存中，但这是非常困难的。一般来说，只有在形成内存访问局部性时才可能做到。也就是说，由于程序一次性访问的内存空间非常小且距离非常近，所以会对一次性读取到缓存的内存空间进行多次读写。

在虚拟机上灵活运用缓存

遗憾的是，从缓存访问的立场来看，图 1-12 那样的语法树解释器是最糟糕的。构成语法树的节点都是一个个单独的结构体，各自的地址不一定邻近，也不会连续。这就导致难以事先将接下来要访问的内存空间读入到缓存中。

这里如果将语法树转换为指令序列，并储存到连续的内存空间上，那么内存访问局部性就会有所增强，性能也会因为缓存的作用而得到极大的提升。

Ruby 1.9 中引入的被称为 YARV 的虚拟机就使用这样的方法实现了性能提升。YARV 是 Yet Another Ruby VM（另一个 Ruby 虚拟机）的缩写。之所以叫这个名字，是因为当初开发时已经有多个以运行 Ruby 为目的的虚拟机在开发了。起初，YARV 只是一个实验项目，但在这些虚拟机中只有它达到了能运行 Ruby 语言全部特性的效果，因此最终 YARV 替代了 Ruby 自己的虚拟机。

虚拟机的优点和缺点

采用虚拟机的语言中最有名的应该是 Java 了吧，但虚拟机这项技术并不是在 Java 中首次出现的，而是在 20 世纪 60 年代后期就已经有了。比如，20 世纪 70 年代初出现的 Smalltalk 语言就因从早期就采用了字节码而名声大噪（这只是部分原因）。再往前说，后来设计了 Pascal 语言的尼古

① 现在的 CPU 都把缓存分为多个层级来增大缓存容量。即便如此，容量还是比主存小得多，而且也没有解决难以事先将接下来要访问的内存空间读入到缓存的问题。

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

拉斯・沃斯（Niklaus Wirth）以 Algol68 语言为基础设计的 Eular 语言据说也完成了虚拟机的实现。Smalltalk 之父艾伦・凯（Alan Kay）说，Smalltalk 的虚拟机的实现受到了 Eular 的虚拟机的启发。

说起 Pascal 就会想起 UCSD Pascal。由加州大学圣地亚哥分校开发的 UCSD Pascal 把 Pascal 程序变更为字节码 P-code 之后运行。将 Pascal 程序变更为 P-code，可以轻松地将 UCSD Pascal 移植到各种操作系统和 CPU 的计算机上，这也使得 UCSD Pascal 作为具有较强移植性的编译器被广泛使用。

从这里我们就能明白，虚拟机最大的优点就是拥有可移植性。配合各种各样的 CPU 生成机器语言的代码生成处理是编译器中最复杂的部分。根据后续出现的各种 CPU 重新开发代码生成处理，对语言处理器的开发者来说是很大的负担。

现在 x86 和 ARM 等架构占据统治地位，CPU 的种类比以往减少了许多，但在 20 世纪六七十年代，新架构层出不穷，甚至同一家公司的同一系列的计算机也会根据型号而使用不同的 CPU。虚拟机在减少这类负担上起到了很大作用。

另外，虚拟机能够配合目标语言进行设计，因此我们就可以将指令集的范围限定在实现这个语言所必需的指令中。与通用 CPU 相比，可以缩小规格，开发也变得更简单。

但虚拟机并非只有优点。与在硬件上直接执行相比，模拟虚拟的 CPU 运行的虚拟机在性能上有很大的问题。采用了虚拟机的语言处理器会产生几倍，甚至几百倍的性能损失。不过我们可以使用 JIT 编译等技术在一定程度上减少这种性能损失。

虚拟机的实现技术

用硬件实现的真正的 CPU 与用软件实现的虚拟机在性能上各有不同。下面我们来看一下虚拟机性能相关的实现技术，以下是具有代表性的几种。

(1) RISC 与 CISC
(2) 栈与寄存器
(3) 指令格式
(4) 直接跳转

RISC 是 Reduced Instruction Set Computer（精简指令集计算机）的缩写，是通过减少指令的种类、简化电路来提高 CPU 性能的架构。在 20 世纪 80 年代流行的架构中，具有代表性的 CPU 有 MIPS 和 SPARC 等。在移动设备上广泛使用的 ARM 处理器就属于 RISC。

CISC 是与 RISC 相对的一个词汇，是 Complex Instruction Set Computer（复杂指令集计算机）的缩写，简单来说就是“不是 RISC 的 CPU”。CISC 的每个指令执行的处理都非常大，而且指令的种类繁多，因此实现起来也比较复杂。

不过，RISC 与 CISC 的对立是 21 世纪之前的事情了，在如今的硬件 CPU 中，RISC 与 CISC 的对立没有任何意义。这是因为纯粹的 RISC 的 CPU 失去了人气，现在已经很少见到了。即便如

此，SPARC 还是存活了下来，被日本超级计算机“京”等设备采用。

RISC 中前景较好的 ARM 也在不断增加指令，朝着 CISC 的方向发展。而作为 CISC 代表架构的英特尔 x86，通过在表面上提供复杂的指令集[①]以维持与过去版本的兼容性，并在内部把指令转换为类 RISC 的内部指令（μop），从而实现了高速运行。

CISC 在虚拟机上有优势

但对虚拟机来说，RISC 和 CISC 之争有不同的意义。如果是用软件实现的虚拟机，我们就不能忽视取指令（Instruction Fetch，IF）处理所需要的成本。也就是说，做同样的处理时所需的指令数越少越好。好的虚拟机指令集是类 CISC 架构的指令集，它的全部指令都是高粒度的。

虚拟机的指令要尽可能地抽象，程序设计得小一些会比较好。有些虚拟机以紧凑化为目标，提供复合指令，把频繁被连续调用的多条指令整合为一条，这样的技术称为“指令融合”或“super operator”。

栈与寄存器

虚拟机架构的两大流派是栈式虚拟机和寄存器式虚拟机。栈式虚拟机原则上通过栈对数据进行操作（图 1-13），而寄存器式虚拟机的指令中包含寄存器编号，原则上对寄存器进行操作（图 1-14）。

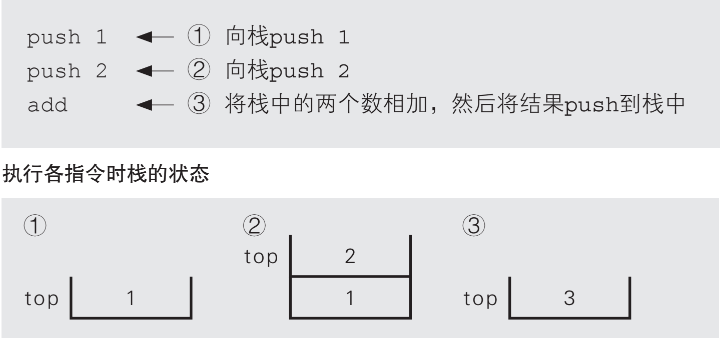


图 1-13　栈式虚拟机的指令及其结构

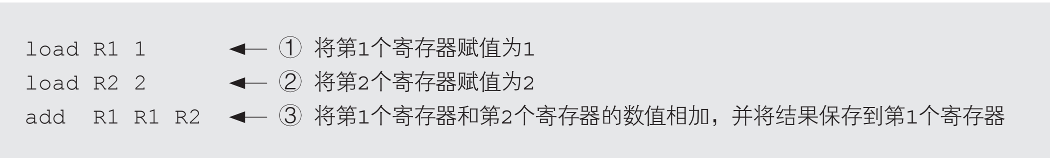


图 1-14　寄存器式虚拟机的指令

与寄存器式虚拟机相比，栈式虚拟机更为简单，程序也相对较小。然而，由于所有的指令都通过栈来交换数据，所以对指令之间的先后顺序有很大的依赖，很难实施交换指令顺序这样的优化。

而寄存器式虚拟机由于指令中包含寄存器信息，所以程序相对较大。这里需要注意的是，程序大小与取指令处理的成本不一定相关，这一点我们在后面也会提到。另外，寄存器式虚拟机由于显式指定了寄存器，所以对指令顺序依赖较小，优化空间较大。不过，小规模语言高度优化的例子几

---

① 前几天有消息称 x86 的 move 指令过于复杂，仅用这个指令就可实现图灵完全。也就是说，理论上仅用 move 指令就能编写出任何算法。

乎不存在，所以这一点也就没那么重要了。

那么栈式虚拟机和寄存器式虚拟机哪个更好呢？这个问题现在还没有定论，使用这两种架构的虚拟机都有很多。表 1-2 展示了这两种架构在各种语言的虚拟机中的使用情况。我们发现，即使是同一语言，也会因实现的不同而采用不同的架构，有时采用栈式虚拟机，有时采用寄存器式虚拟机。这种现象很有趣。

表 1-2　各种语言的虚拟机架构

语言	虚拟机	架构
Java	JVM	栈式虚拟机
Java	Dalvik（Android）	寄存器式虚拟机
Ruby	YARV（Ruby 1.9 之后的版本）	栈式虚拟机
Ruby	mruby	寄存器式虚拟机
Lua	lua	寄存器式虚拟机
Python	CPython	栈式虚拟机

指令格式

Smalltalk 出现之后，虚拟机解释的机器语言（指令序列）就开始被称为字节码了。这是因为 Smalltalk 的指令是以字节为单位的。后来“字节码”这个单词被继承了字节单位这一特性的 Java 发扬光大。

不过，不是所有虚拟机都拥有字节单位的指令集，例如 YARV 和 mruby 的指令集就是用 32 位整数表示的。对很多 CPU 来说，32 位整数是最容易处理的长度，多被称为“字”（word），所以这些指令序列的学名叫“字码”可能更为合适。但是“字码”这个词不仅不好读，还不容易让人理解，所以完全没有得到普及，以至于人们慢慢地就放弃了，有时就直接管它叫字节码了。

字码的优缺点

字节码与字码都有各自的优缺点。与每个指令必定消耗 32 位的字码相比，字节码的程序更加紧凑。另一方面，由于字节码中的 1 个字节相当于 8 位，只能表示 256 个状态，所以操作数（指令的参数）只能保存在指令之后的字节中，这样就会增加从指令序列中取出数据的取指令次数。前面也说过，在用软件实现的虚拟机中，取指令处理的成本较高，因此字码在性能上更有优势。

另外，字码在“地址对齐”这一点上也有优势。在一些 CPU 中，地址如果不是特定数的倍数，直接对其进行访问就会出错。在这种情况下就需要从已对齐的（地址统一为特定数的倍数）地址中取出数据，将偏移的部分切取出来。即便访问不会出错，成倍数的地址与不成倍数的地址（因为在内部进行了前文所述的切取等）在访问速度上也会有很大差别。

地址为 2 的倍数称为 16 位对齐，为 4 的倍数称为 32 位对齐。字码中所有的指令都必须符合地址对齐这一标准，字节码则并非如此。根据 CPU 种类和地址状态的不同，有时字节码平均每个指令的取指令成本会很高。

总的来说，字节码的指令序列相对较短，在内存使用量上有优势，但从取指令的次数和所需时间等性能方面来看，字码更有优势。

看一下 mruby 的指令

接下来我们看一下虚拟机指令的实际例子，比如图 1-15 中的 mruby 指令。

mruby 的指令通过末尾的 7 位来确定指令种类。通过 7 位来确定指令种类，这就意味着最多可以实现 128 种指令。实际上，包括预备的 5 种指令在内，mruby 共准备了 81 种指令。

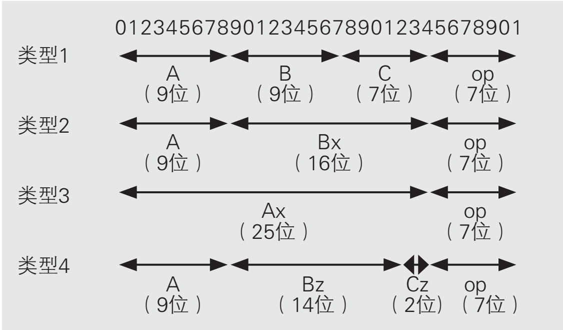


图 1-15 mruby 的指令结构

指令长度共 32 位，其中 7 位用于确定指令种类，这就表示剩余的 25 位可用于操作数。mruby 的指令可根据操作数部分的使用方法（划分方法）划分为 4 种类型。

类型 1：3 个操作数

指令类型 1 包含 A、B、C 这 3 个操作数。A 是 9 位，B 也是 9 位，C 是 7 位。也就是说，操作数 A 和 B 的最大值是 511，操作数 C 的最大值是 127。操作数 A 和 B 多用于指定寄存器。例如，寄存器之间的移动指令 OP_MOVE 在此类型中的命令为

```
OP_MOVE A B
```

这表示把操作数 B 指定的寄存器的内容复制到操作数 A 指定的寄存器上。OP_MOVE 指令不使用操作数 C。

使用操作数 C 的指令，例如有调用方法的 OP_SEND。

```
OP_SEND A B C
```

这表示调用操作数 A 指定的寄存器（这里称为寄存器 A）中保存的对象中通过操作数 B 指定的符号[①]（准确来说是符号表中第 B 个符号）所代表的方法。范围在 A + 1 到 A + 1 + C 的寄存器的值是方法的参数，方法调用的返回值保存到寄存器 A 中。

正如刚才介绍的 OP_MOVE 指令那样，有些操作数在类型 1 的几个指令中都没有用到。虽然这部分空间被浪费掉了，但是从访问的便捷性和效率来考虑，这种情况还是可以接受的。

类型 2：2 个操作数

指令类型 2 中没有操作数 B 和 C，取而代之的是一个大的（16 位）操作数。这个操作数分为无符号数（Bx）和有符号数（sBx），根据指令的不同区分使用。使用 Bx 的有 OP_GETIV 等指令，

① 符号（symbol）是指语言处理器在内部识别方法名时使用的值，不同的字符串会被分配不同的值。

如下所示。

```
OP_GETIV A Bx
```

这表示将符号表中第 Bx 个符号指定的 self 实例变量保存在寄存器 A 中。

使用 sBx 的指令有跳转命令，形式如下。

```
OP_JMP sBx
```

这个指令可以使下一个指令的地址由现在的地址跳转到偏移 sBx 个位置的地方。sBx 是有符号数，因此前方后方都可以跳转。OP_JMP 指令不使用操作数 A。使用操作数 A 的有条件跳转的指令例子如下所示。

```
OP_JMPIF A sBx
```

这表示在寄存器 A 为真的情况下，跳转 sBx 个位置。

类型 3：1 个操作数

指令类型 3 把操作数部分整合为 1 个 25 位的操作数（Ax）进行处理。类型 3 的指令只有 OP_ENTER。

```
OP_ENTER Ax
```

OP_ENTER 根据 Ax 指定的位模式进行方法的参数检查。OP_ENTER 将 25 位中的 23 位分割为 5/5/1/5/5/1/1 来解释参数，每位的含义如表 1-3 所示。

表 1-3　OP_ENTER 的参数指定

位	内容
5	必需参数的数量
5	可选参数的数量
1	是否有 rest 参数（* 参数）
5	末尾的必需参数的数量
5	关键字参数的数量（暂未使用）
1	是否有关键字 rest 参数（** 参数）（暂未使用）
1	是否有块（block）参数

从头开始分割 25 位的 Ax 操作数

类型 4：类型 1 的变形

指令类型 4 把 B 和 C 操作数的部分（16 位）分割为 14 位的 Bz 操作数和 2 位的 Cz 操作数。指令类型 4 的指令只有 OP_LAMBDA。

mruby 准备了从指令中获取操作数的宏，使用这些宏就可以从指令（的字）中获取操作数。这些宏不会进行指令类型的检查，所以开发者要注意正确使用宏。获取 mruby 指令的操作数的宏如表 1-4 所示。

表 1-4 获取 mruby 指令的操作数的宏

宏名称	含义
GET_OPCODE(i)	获取指令的类别
GETARG_A(i)	A 操作数（9 位）
GETARG_B(i)	B 操作数（9 位）
GETARG_C(i)	C 操作数（7 位）
GETARG_Bx(i)	Bx 操作数（16 位）
GETARG_sBx(i)	sBx 操作数（有符号型 16 位）
GETARG_Ax(i)	Ax 操作数（25 位）
GETARG_b(i)	Bz 操作数（14 位）
GETARG_c(i)	Cz 操作数（2 位）

解析循环

如果可以使用这样的结构将源代码转换为虚拟机的指令序列，就可以轻松实现虚拟机的基本结构。

虚拟机的中心部分，也就是解析循环（interpreter loop），用伪代码表示时如图 1-16 所示。

是不是简单到让你吃惊？即使指令增加，也只是 switch 语句的 case 增加了而已。

不过，就算基本结构很容易实现，要实现具有实用性的语言还是有很多事情需要考虑，比如这里没有提到的怎样实现运行时栈、如何构建方法调用和异常处理的机制等。由此我们也能看出，理论和实践之间还隔着一条巨大的鸿沟。

```
typedef uint32_t code;

int
vm_loop(code *pc)
{
  code i;

  for (;;) {
    switch (GET_OPCODE((i = *pc))) {
    case OP_MOVE:
      stack[GETARG_A(i)] = stack[GETARG_B(i)];
      break
    case OP_SEND:
      ...
      break;
      ...
    }
  }
}
```

图 1-16 虚拟机的基本结构（使用 switch 语句）

直接跳转

在很多情况下，实用的虚拟机都是速度优先的，因此我们也想提高解析循环的效率。提高虚拟机解析循环效率的技术中比较有名的是直接跳转（direct threading），其中使用了 GCC（GNU Compiler Collection）的扩展特性。

GCC 中可以获取标签（label）的地址并跳转到这个地址。标签的地址可以通过“&& 标签名”

获取，跳转到标签的方法是“goto *标签”。使用此项功能，我们就可以使用跳转代替 switch 语句来构建虚拟机。

使用直接跳转实现解析循环的代码如图 1-17 所示。

```
typedef uint32_t code;

#define NEXT i=*++pc; goto *optable[GET_OPCODE(i)]
#define JUMP i=*pc; goto *optable[GET_OPCODE(i)]

int
vm_loop(code *pc)
{
  code i;
  /* 按指令编号顺序排列的标签地址 */
  static void *optable[] = {
    &&L_OP_MOVE, &&L_OP_SEND, ...
  };

  JUMP;

L_OP_MOVE:
  stack[GETARG_A(i)] = stack[GETARG_B(i)];
  NEXT;
L_OP_SEND:
  ...
  NEXT;
  ...
}
```

图 1-17　使用直接跳转的情况

实际上包括 mruby 在内，使用直接跳转的虚拟机的实现中基本上都提供了编译选项，供用户选择是使用 switch 语句还是使用直接跳转。这是因为标签地址的获取只是 GCC 的扩展特性，不能保证一直可用。使用切换宏实现循环的代码如图 1-18 所示。

```
typedef uint32_t code;

/* 只支持带GCC扩展功能的编译器 */
#if defined __GNUC__ || defined __clang__ || defined __INTEL_COMPILER
#define DIRECT_THREADED
```

```
#endif

#ifdef DIRECT_THREADED

#define INIT_DISPATCH JUMP;
#define CASE(op) L_ ## op:
#define NEXT i=*++pc; goto *optable[GET_OPCODE(i)]
#define JUMP i=*pc; goto *optable[GET_OPCODE(i)]
#define END_DISPATCH

#else

#define INIT_DISPATCH for (;;) { i = *pc; switch (GET_OPCODE(i)) {
#define CASE(op) case op:
#define NEXT pc++; break
#define JUMP break
#define END_DISPATCH }}

#endif

int
vm_loop(code *pc)
{
  code i;
#ifdef DIRECT_THREADED
  static void *optable[] = {
    &&L_OP_MOVE, &&L_OP_SEND, ...
  };
#endif

  INIT_DISPATCH {
    CASE(OP_MOVE) {
      stack[GETARG_A(i)] = stack[GETARG_B(i)];
    }
    NEXT;
    CASE(OP_SEND) {
      ...
    }
    NEXT;
    ...
  }
```

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

```
  END_DISPATCH;
}
```

图 1-18 使用切换宏的情况

使用这项技术，即使在没有 GCC 扩展特性的编译器上，也可以用 `switch` 语句得到相应的速度。而在有 GCC 扩展特性的编译器上，则可以使用直接跳转技术实现速度更快的虚拟机。

小结

本节讲解了运行时的核心部分——虚拟机的实现，至此语言处理器的基础部分就粗略地讲解完了。下个月[①]开始我会把讲解的重心放在语言设计上。

时光机专栏

虽然我也想在Streem语言中使用虚拟机……

本节是 2014 年 6 月刊中刊登的内容。接着上一节对 yacc 的介绍，这里讲解了虚拟机的实现。讲解时使用了 mruby 作为示例，是因为过于简单的例子不容易让大家把握虚拟机实现的整体情况。最重要的原因是，我打算在 mruby 的虚拟机的基础上实现其他语言（今后要去实现的语言）的虚拟机。

实际上 Streem 的实现采用了直接遍历语法树这种简单的解释器，所以本节讲解的内容对于 Streem 来说不会起到任何作用，但对于虚拟机的实现还是有价值的，因此本书选择保留这部分内容。虽然我也打算把 Streem 的简单的解释器替换为本节介绍的虚拟机，但却苦于没有时间。时间管理成为我最大的障碍，这种情况已经不是一次两次了……

① 本书是由杂志连载内容整理而来的，因此有“下个月”之说。——译者注

1-4 编程语言设计入门（前篇）

关于语言的实现我们已经有了大致的了解，接下来就来思考一下语言的设计吧。作为案例，本节我们将回顾一下Ruby早期的设计。Ruby是作为一门支持脚本编程的面向对象语言开发的。

假设你想创造一门新的编程语言，并且不是玩玩看的心态，而是希望它有朝一日能成为在全世界广泛使用的“人气语言”。那么，你该如何做呢？

创造人气语言的方法

比起性能和功能，语言规范更能决定一门新的编程语言的人气。然而，基本上没有哪本书或哪个网页会告诉你如何设计一门语言。

不过仔细想想，也几乎没有什么人设计过正经的语言。市面上虽然有自制编程语言相关的教材，但是这些教材介绍的都是编程语言的实现方法，里面介绍的语言也不过是一些例子而已，大多是现有语言或现有语言的子集。语言的设计则在这些教材的考虑范围之外，可能连编写这些教材的人也没有设计人气语言的经验。

的确如此。没有多少编程语言能达到在世界上被广泛使用的程度。即使把历史上所有的人气语言全算上，恐怕也不到几百种。当然，这也要看怎么定义“人气”这个词了。也就是说，这些语言的设计者在全世界也不过几百人，而且其中一些人已经不在了。

作为为数不多的语言设计者的一员，我觉得我有使命向大家介绍语言设计的秘诀。本书真正的目的也在于此。

心里的疑问

有志成为语言设计者的人在开始设计新的语言时，脑海中经常会掠过以下疑问。

- 真的需要新的语言吗
- 这门语言是做什么的
- 目标用户是谁
- 采用什么样的功能

我们没有必要为这些疑问而烦恼，因为即使你为此烦恼，对你设计一门好的语言也毫无益处。

不过，这里我们还是来思考一下这些问题。比如第一个问题，其实只要是图灵完全的语言，就可以用来编写所有算法。现有的编程语言都已经证明是图灵完全的，所以从软件开发（ = 编写算法）的角度来看，完全不需要新的语言。

然而，现实是在过去五十多年不断有新的语言被创造出来，这并不是因为已有的语言不能编写某个算法，而是因为用新的语言编写起来更方便或者写起来更爽。而你之所以会产生是否真的需要新语言的疑问，恰恰就是因为你的心底已经有了创造一门语言的想法。既然有了这样的想法，就无须为“是否有必要”这种问题而烦恼了。

自己想用就足够了

对于“这门语言是做什么的”和“目标用户是谁”的问题，我觉得有必要做一下补充。

作为资深的编程语言迷，我学习了很多编程语言。在 Ruby 成名之后，我与很多语言设计者也都进行过交流，比如 C++ 的设计者本贾尼・斯特劳斯特卢普（Bjarne Stroustrup）、Perl 的设计者拉里・沃尔（Larry Wall）、Python 的设计者吉多・范罗苏姆（Guido van Rossum）和 PHP 的设计者拉斯马斯・勒德尔夫（Rasmus Lerdorf）等。从和他们的交流中我总结出一点，那就是除了设计者本人以自用为目的设计的语言以外，其余的语言大多没有流行起来。

如果连自己都不打算用，在设计时就考虑不到细节，也无法保持激情去将自己设计的语言培养成人气语言。不少语言都是经过十年以上的时间才变得有人气，因此，要想创造一门人气语言，考虑细节和保持激情不可或缺。也就是说，人气语言的目标用户首先是设计者本人，然后才是拥有相似特质的用户。而“这门语言是做什么的”则取决于设计者本人想做什么。

决定了目标用户和语言用途之后，就没有必要为最后一个问题，也就是“采用什么样的功能”而烦恼了。不过这里面也隐含着一些诀窍，之后我们再进行说明。

■ 开发 Ruby 的契机

漂亮话说再多也没有说服力，这里我们来看一下 Ruby 的案例。我与 Ruby 打交道了二十多年，可以说的东西有很多。这里我们重点回顾一下决定语言设计方向的开发初期。

先从 Ruby 的开发背景说起。

Ruby 的开发始于 1993 年。我对编程语言产生兴趣是在 20 世纪 80 年代初，当时我还在鸟取县读高中，从那个时候起我便对 Pascal、Lisp 和 Smalltalk 等编程语言产生了浓厚的兴趣。

那时我没有自己的计算机，还不能自由地编写程序，但不知道为什么就对编程语言产生了兴趣，真是不可思议。比起写什么程序，编程语言这一编写程序的手段对我的吸引力更大。

但是，因为我住在乡下，找不到什么资料或者文献来学习，所以吃了不少苦头。那个时候互联网还没有普及，学校图书馆里也基本没有计算机相关的书，这让我头疼不已。

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

为了获得编程语言的相关信息，我只好在计算机杂志上找编程语言的相关内容，或者去附近的书店看一些类似于大学教材的书（书很贵，当时买不起），所以我一直很感激当时经常去的那家书店。

后来我上了大学，图书馆里摆满了各种图书、杂志和论文，非常齐全，当时就觉得自己生活在了天堂。我就是这样掌握了编程语言的相关知识，这些知识在我后来的语言设计中也起到了非常大的作用。就像没有不读书的作家、没有不了解旧棋谱的职业棋手一样，在设计新的语言时，广泛了解现有语言的相关知识是很重要的。

1993 年有很多空闲时间

时间到了 1993 年。那时我已经大学毕业，成为了一名职业程序员，工作就是根据公司的业务要求开发软件。在那之前我开发了公司使用的内部系统，还在 UNIX 工作站上开发了桌面以及可以添加附件的邮件系统等。如今在 Windows 和 Mac 系统上这没有什么稀奇的，但在当时的 UNIX 工作站上却没有这样的系统。即使有类似的也基本不支持日语，所以只能自己开发。

但是在泡沫经济破灭之后，公司整体就变得不景气了，内部系统又不能带来经济效益，于是公司决定停止新功能开发，继续使用已经开发完成的功能。

开发团队被解散，只有少数人作为维护人员留了下来。不知是幸还是不幸，我也是这些少数人之一。但是因为已经停止了开发，所以我也没什么事情可做。偶尔有人打过来电话说计算机无法正常运行，我也只要回复“请重启一下”就行。那段日子就是这么过来的，完全是在坐冷板凳。

图书的策划成为契机

不过，这也不全是坏事情。虽然公司不景气，我不用怎么加班[①]，而且也没有了奖金，与泡沫经济时期相比收入减少了很多（当时刚结婚的我手头比较紧），但幸运的是我没有被开除，所以也不用去找工作。眼前有计算机，事情少而且不重要，所以也没人管。时间和精力都很充沛，就开始想去做点什么了。那段时间开发了几个实用的小程序，后来因为一个偶然的契机，我决定去实现埋藏在心中多年的一个梦想——创造一门编程语言。

这个“偶然的契机”是这样的。当时和我同部门的一位前辈策划出一本书，在开始动笔时他找我商量：“我决定写一本通过创建编程语言来学习面向对象的书，你可以帮我写编程语言的部分吗？”

作为编程语言迷的我对这个策划内容非常感兴趣，于是就答应了。但这个策划最终没能通过编辑会议的评审，很快就流产了。创造一门编程语言是我多年来的一个梦想，我好不容易鼓起了干劲，不想就这样停止。以前只是徒有梦想，想象不出语言完成时是什么样子，所以一直没有动力去做，现在好不容易燃起了激情，就此停止就太可惜了。

① 通常日本公司都会有加班费。——译者注

正是这股“干劲”开启了 Ruby 二十年的历史。当时，我做梦都没想到 Ruby 能成长为一个被全世界广泛使用的语言。

■ Ruby 早期的设计

前面我们已经探讨过了在打算创造一门新语言时脑海中会涌现的疑问，虽然在二十年后的今天我可以明确地说自己已经对这些问题不在意了，但当时的我很年轻，还是稍微犹豫了一下。在经过短暂的思考之后，我决定创造属于自己的语言。现在想想，就是当时的这个选择决定了后来的一切。

那时我是 C 程序员，多使用 C 和 shell 脚本语言。工作中中等规模以上的系统用 C 来开发，而日常使用的比较小规模的程序则用 shell 脚本开发。当时（实际上现在也是）我既没有对 C 感到不满意，也没有觉得创造一门给 C 增加面向对象功能的新语言有什么吸引力。这可能是因为当时已经有了 C++，还有我在大学毕业设计中设计过一门以 C 为基础的面向对象语言（虽然没有达到令自己满意的程度）。

对 shell 脚本不满意

我反而对 shell 脚本不是很满意。当时我使用的是 bash，如果仅仅是排列一下命令行，再加上一些简单的控制结构的话，那么用这种简单的语言也就足够了。但是，随着程序不断完善而逐渐变得复杂，就容易出现连自己都看不懂的情况，这让我觉得不太满意。另外，shell 脚本没有正规的数据结构，这一点也让我感到不满。总之，shell 脚本只是加了些逻辑控制的命令行输入，说到底也不过是个“简易语言”，这正是它的问题所在。

当时在与 shell 脚本相近的领域（脚本语言领域）里有更接近普通语言的 Perl 语言，但是在我看来，Perl 语言也带着一种“简易语言”的感觉。对于 Perl 只有标量（字符串和数值）、数组和散列这几种数据结构，我也很不满。因为这样就无法直接表达一些复杂的数据结构了。

当时还是 Perl 4 的时代，Perl 5 的面向对象功能只不过是坊间传闻，但是传闻中的 Perl 5 的面向对象功能听上去也不是很让人满意。我觉得相较于 Perl，拥有更丰富的数据结构的语言会更好。另外，我从高中时就开始痴迷面向对象编程，所以我希望编程语言不仅能够处理结构体，还能够真正地支持面向对象编程。

Python 过于普通

另外，还有一门叫作 Python 的语言。那个时候关于 Python 的信息还很少，我下了很多功夫去研究，结果发现面向对象功能是后来加上去的，而且感觉这门语言过于普通，所以我不太喜欢。我也知道自己的想法是多么地自大，但只要一说起“理想的语言”这个话题，我这个编程语言迷的话匣子就关不上了。

可能有人会问“过于普通”是什么意思。这是说 Python 在语言层面上不支持正则表达式，字符串操

作功能也不够强大，让人感觉不到它在语言层面上支持脚本开发（这里指的是 20 年前的 Python）。

通过缩进来表示代码块是 Python 的特征之一，这是一个很有意思的尝试，但同时也是它的一个缺点。比如，当你想根据模版自动生成代码时，如果不能保持正确的缩进，程序就不能正常工作；由于代码块是通过缩进来表示的，所以在语言层面上需要明确区分表达式和语句，等等。

这样说来，Python 与普通的 Lisp 方言相比，除了语法更容易理解一些之外，似乎也没有什么区别。现在想想，我完全忽视了社区和类库的存在，但当时我还没有认识到它们的重要性。

让脚本语言支持面向对象

不过，通过考察其他语言，我清楚自己想做什么样的语言了，那就是一个类似于 shell 脚本、比 Perl 更加接近普通语言、可以自己定义数据结构并具有面向对象功能的语言。当然，这门语言要比 Python 更能无缝地进行面向对象编程，而且还必须支持包括字符串操作在内的脚本编程所需的特色功能，以及具备库。

近年来，脚本编程变得越来越重要，Perl 和 Python 的出现频率也越来越高，然而同在脚本编程领域的面向对象编程的必要性却没有得到足够的认识。

当时人们一般认为面向对象语言是仅在大学研究或大规模的复杂系统开发中使用的技术，而不会在脚本编程这种小规模的简单编程中使用。不过，这种情况总算有了转变的苗头。

Perl 终于计划在今后支持面向对象功能。Python 虽然已经是面向对象语言，但是这个功能是后来增加的，所以（当时）并非所有的数据都是对象。当我想去编写一门面向对象语言的时候，Python 已经成了支持面向对象编程的过程式编程语言。如果这时出现一门以脚本编程为主、支持过程式编程的面向对象语言，那么它一定非常好用，至少我自己很乐意去使用。

说着说着干劲就来了。程序员三大美德[①]之一的傲慢在我身上体现得淋漓尽致，我决定，既然要做，就要做出不输给 Perl 和 Python 的东西来。盲目自信是可怕的，但往往这样的自信会成为动力的源泉。

■ 开始开发 Ruby

于是我开始了 Ruby 的开发。最开始决定的是名字，名字很重要。Perl 的名字源于“珍珠”（pearl）这个单词，于是我决定仿效 Perl，为这门语言选一个宝石的名字。宝石的名字大多比较长，比如 Diamond 和 Emerald，我一直找不到合适的，挑来挑去最后只剩下了 Coral（珊瑚）和 Ruby（红宝石）。Ruby 这个名字既短又美，于是我最终选择了它。那个时候没怎么细想，不过因为编程语言的名字经常被人叫起，所以最好既好读又让人印象深刻。

如果大家决定开发自己的语言，就一定要多花精力想一个好名字。能够清晰地表达出语言特

① Perl 的设计者拉里・沃尔说程序员有三大美德，分别是懒惰、急躁和傲慢。当然，普通情况下不会称这些特质为美德。

征的名字是最好的，不过像 Ruby 这种与语言特征完全无关的名字也可以。最近出现了常用名字的“googleability”（可搜索性）很低的问题。这个问题在 1993 年 Ruby 开始开发的时候还不存在。

对代码块结构的表现方式的思索

接着决定的是使用 end 关键字表示代码块。C、C++ 和 Java 在代码块里都使用大括号（{ }）来括住多条语句，这么做会出现一个问题，那就是把单条语句变为多条语句时容易忘记加大括号（图 1-19）。尽管 Pascal 用 begin 和 end 代替了大括号，但因为也有单条语句和多条语句的区别，所以也存在同样的问题。

我不喜欢这种单条语句和多条语句的问题，所以想在自己的语言中杜绝这种问题的发生，实现这个目标的方法有三种。

(1) 单条语句中不允许省略大括号的 Perl 方式
(2) 用缩进表示代码块的 Python 方式
(3) 不区分单条语句和多条语句，用 end 结束代码块的 Eiffel 方式（图 1-20）

```
// 多条语句时用大括号括起来
if (cond) {
  statement1();
  statement2();
}

// 单条语句时也可以不使用大括号
if (cond)
  statement1();

// 把单条语句变为多条语句时忘记加大括号
if (cond)
  statement1();
  statement2();  // 不出现语法错误
```

图 1-19　单条语句和多条语句的问题

```
■ 单条语句的情况
if cond
  statemen1();
end

■ 多条语句的情况（与单条语句无区别）
if cond
  statemen1();
  statemen2();
end

■ 有多个代码块时像梳子一样
if cond
  statemen1();
elsif cond2
  statemen2()
else
  statemen3()
end
```

图 1-20　Eiffel 方式（梳子型代码块结构）

自动缩进的课题

多年来我一直使用 Emacs 文本编辑器，非常熟悉它的语言模式，而且最喜欢这个语言模式提供的自动缩进功能。输入一些代码后，编辑器就会自动帮你缩进，这种感觉就像是和编辑器合力编写代码一样。

在 (2) 的 Python 方式中，缩进本身是用来表示代码块结构的，因此没有自动缩进的余地（不过，在行的末尾输入冒号，缩进会更加深入）。另外，使用缩进表示代码块的 Python 中明确区分了语句和表达式，由于我受不区分语句和表达式的 Lisp 的影响较大，所以对这一点不是很喜欢。因此，我最终没有采用这种用缩进表示代码块的 Python 方式。

上学时 Eiffel 给了我很大影响，那时我读了一本名为《面向对象软件构造》的书，受其影响，我设计了一门语义上类似于 Eiffel（但是语法类似于 C 语言）的语言作为毕业设计。尽管不能说这个尝试取得了成功，但接下来我准备在语法上（而非语义上）借鉴 Eiffel，看看效果如何。

自制 Emacs 的语言模式

这里让我担心的依旧是自动缩进功能。在当时的 Emacs 语言模式中，主流做法是像 C 那样用符号标记代码块，以此进行自动缩进，而 Pascal 等使用关键字表示代码块的语言的模式则是用快捷键来增加或减少缩进，这样就没有了自动缩进的畅快感。

于是我花了几天时间与 Emacs Lisp 展开搏斗，使用正则表达式对 Ruby 语法进行了简单的分析，创建了在使用 `end` 的语法中也可以自动缩进的 Ruby 语言模式的模型，由此也证明在使用 `end` 的、语法类似于 Eiffel 的语言中也可以实现自动缩进功能。这样一来，我就可以放心地在 Ruby 的语法中使用 `end` 了。反过来说，如果当时没有成功开发出可以自动缩进的 Ruby 语言模式，那么 Ruby 语法也就不是现在的样子了。

在设计上选择使用 `end` 的代码块结构还有一个预料之外的好处。因为 Ruby 的很大一部分是使用 C 实现的，所以就必然需要区别使用 C 和 Ruby。不过 C 和 Ruby 的代码风格完全不同，所以可以一眼看出当前是在用哪种语言工作，这就降低了大脑的模式切换成本。虽然这个成本微不足道，但是它对保持良好的编程劲头还是非常有好处的。另外，今后当 Perl、Python 和 Ruby 被当成脚本语言的竞争对手时，我想每种语言都拥有不同的代码块构造（Perl 是大括号，Python 是缩进，Ruby 是 `end`）或许能帮助它们继续生存下去。

是 else if 还是 elsif 还是 elif

说点题外话，如果使用了上述解决多条语句问题的方法，就不能像 C 那样编写 `else if` 语句了。因为 C 的 `else if` 会被解释为在 `else` 后面紧跟着一个无大括号的单条 `if` 语句（图 1-21）。用 Ruby 的语法编写 `else if` 语句，代码如图 1-22 所示。

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

```
// (a) 使用了else if的以下语句
if (cond) {
  ...
}
else if (cond2) {
  ...
}

// 如果不省略大括号，就会变成这样
if (cond) {
  ...
}
else {
  if (cond2) {
    ...
  }
}
```

图 1-21　C 的 else if

```
# 总之，如果Ruby没有用elsif
# 就需要写成下面这样
if cond
  ...
else
  if cond2
    ...
  end
end

# 还是用elsif更好
if cond
  ...
elsif cond2
  ...
end
```

图 1-22　Ruby 的 else if

从图 1-22 的代码来看，还是用 `elsif` 比较好。顺便说一句，Perl 和 Ruby 用的是 `elsif`，而 shell 脚本和 Python（还有 C 预处理器）用的是 `elif`。这个差别真是有意思。

据说 Python 是从 shell 脚本和 C 预处理器那里继承的 `elif` 这一写法，而 shell 脚本等又是从古老的 Algol 系列继承而来的。此外，像 shell 脚本的 `fi` 和 `esac` 那样将表示开始的关键字倒着拼写来表示结束，据说也是起源于此。

很遗憾我不知道 Perl 为什么用了 `elsif`，但 Ruby 是因为以下两点。

- `elsif` 与 `else if` 发音相同而且长度较短（`elseif` 长一些，而 elif 的发音发生了改变）
- Ruby 在基本语法上借鉴最多的语言 Eiffel 用的也是 `elsif`

一个关键字也是有历史原因的。

再扯得远一些，Perl 的语法虽然跟 C 基本相同，但因为不能省略大括号，所以基于图 1-21 的原因不支持 `else if`。不过，如果在语法上明确加入 `else` 和 `if` 的组合，兴许也能支持 `else if`。在很久之前，有一天我一时兴起改了一下 Perl 的源代码，没想到只花几分钟稍微修改了一下 yacc 描述，就做出了支持 `else if` 的 Perl。不知道 Perl 社区的人为什么至今还没有动手去做，真是让人费解。

开始实现

确定了基本方针和语法的方向之后，接下来就到了实现环节。幸好我手上还有以前随便做的“小儿科”语言的源代码，所以就决定以这个为基础进行开发。

Ruby 的开发始于 1993 年 2 月，之后大致完成了语法分析器和运行时的基础部分，并在半年后的 8 月份开始运行了最早的 Ruby 程序（一个 `Hello World` 程序）。

老实说，那段时期是整个 Ruby 开发过程中最艰难的一段时间。程序员只有看到自己写的代码正常运行起来才能感受到编程的喜悦，而那个时期 Ruby 没有任何可以运行的东西，写来写去也达不到可运行的状态，以至于我几乎没有了支撑下去的动力。

虽说写了语法分析器，但它能做的也只是语法检查。要想运行程序，还需要字符串类，因为 `"Hello World"` 是字符串对象。要编写字符串类，就需要有以 Object 为顶点的面向对象系统，而输出字符串又需要管理 IO 的对象，像这样，需要的东西一个接一个地增加。充分具备程序员三大美德之一的“急躁”的我居然能忍耐那半年，简直是一个奇迹。

人气在于细节

虽说实现了 `Hello World` 的输出程序，但光凭这一点 Ruby 还不能算得上一个可用之物，至此所实现的内容也只是停留在教科书的示例程度。要想达到人气语言这一目标，接下来才是重点。

如何给语言加上自己的特性？如何招揽人气？ Ruby 早期的设计是如何考虑的？我要讲的东西还有很多很多。

不过，“叙旧”叙得有点久，这次的篇幅已经用完了。1-5 节将会继续本节的内容，为大家介绍 Ruby 设计的案例学习的后半部分，敬请期待。

时光机专栏

了解一下常见语言的历史吧

本节是 2014 年 7 月刊中刊登的内容。这里终于开始了对语言设计相关内容的介绍，讲述了 Ruby 语言的开发背景以及历史经过，回答了“为什么想要去做”“在哪些地方遭遇了挫折”“为什么采用这样的语法”等问题。

虽然都是很久以前的事情，但实际上很少有人能讲出常见语言的背景以及隐藏在各种设计背后的理由，所以我认为这一节和下一节是本书的一大亮点。

但话说回来，这些内容本身不过是一些没有用处的知识而已。为了后来人，我真心希望大家能从这些过去的事情中吸取一些教训，比如：

- 设计即决定
- 即使是像语法这样基本的东西，也有各种需要考虑的地方
- 不仔细考虑的话设计就会出错
- 即使仔细考虑也有可能犯错

1-5 编程语言设计入门（后篇）

1-4节讲述了Ruby的诞生，本节将接着上一节的内容，继续讲述Ruby语言设计的相关内容，介绍变量名的命名方法、继承的思考方式、错误处理以及迭代器等是如何确定的，并从中总结语言设计的窍门。

在前面的内容中，Ruby 确定了基本的语法结构，作为编程语言迈出了第一步，但如果只是这样，它也不过是一个随处可见的平庸的语言。现在 Ruby 在语法上只确定了代码块用“`do~end`”括起来、用 `elsif` 实现 `else if` 这几点，接下来还需要在细节上加以完善。

设计原则

在这个阶段，我心目中的 Ruby 除了要满足“成为面向对象语言”这个功能方面的要求以外，还要实现以下几个目标。

- 脱离简易语言的范畴
- 易写易读
- 简洁

“脱离简易语言的范畴”是指在语言规范上不草率了事。当时，特别是在脚本语言领域，很多语言都把完成工作放在第一位，而（貌似）在语言规范上草率了事。比如，明明没什么必要，却以容易实现为由给变量名加上符号，或者用户自定义函数与内置函数的调用方法不同等。

“易写易读”这一点比较抽象。程序不是写一次就结束的，而是要在调试等的过程中反复琢磨，反复修改。对于相同的操作，代码的规模越小越容易理解，所以简洁的代码是最为理想的，不过也不能过于简洁。

世界上也有一些异常简洁的语言，但在事后回过头来看用这些语言写的代码时，则往往无法理解代码的意思，这样的语言通常称为“Write Once Language”，意思是写好之后就不管了。在使用这种语言的情况下，重新解读代码往往要比从头再写一遍更费工夫。只有通过平衡取舍，才能达到易写易读的效果，语言的设计一直都是如此。

另外，在写代码时，如果被迫编写一些在本质上与想做的事情无关的东西，哪怕只是一点点，也会让人感到不快，相信大家都会有这样的想法。这是因为开发时自己只想把精力集中在软件应该用在什么地方这种本质问题上。在不影响理解的前提下，尽量砍掉与本质无关的东西，使实现变简洁，这才是我们希望看到的。

变量名

Perl 是 Ruby 开发初期参考的语言之一。Perl 的变量名开头带有符号，其含义如表 1-5 所示。

表 1-5　Perl 的变量名规则

变量名	含义
$foo	标量（字符串或数值）
@foo	数组（标量数组）
%foo	散列（关联数组）
$foo[0]	访问数组元素
$foo{n}	访问散列元素

其中比较有趣的是访问数组的方式。虽然取数组（@foo）的第 0 个元素，但符号用的却是 $，是如此。也就是说，开头的符号代表了这个变量表达式）的类型。这是因为 Perl 曾是一种通过变明示数据类型的静态类型语言（让人惊讶）。

后来，Perl 引入了引用的概念，这使得包括数组和散列在内的所有东西都可以作为标量来表示。因此，这个静态类型的原则就变得没有那么重要了。

表 1-6　Ruby 的变量名规则

种类	符号	示例
全局变量	$	$foo
实例变量	@	@foo
局部变量	小写字母	foo
常量	大写字母	Foo

但是在看到变量名时，我们最想知道的不是这个变量的类型而是作用域。有些语言（比如 C++）的编码规则要求全局变量或者成员变量前面要有特定的前缀。而在变量名中加入类型信息的编码规则，比如以前美国微软公司经常使用的匈牙利命名法，最近已经完全看不到了。这就说明明示类型信息已经没有必要了。

于是 Ruby 在变量名中增加了表示作用域的符号（表 1-6），比如 $ 是全局变量，@ 是实例变量。然而，如果最常用的局部变量和常量（类名等）也加上符号，就会重蹈 Perl 的覆辙。

让局部变量变得更简洁

经过再三考虑，我决定把规则定为局部变量前面使用小写字母，常量前面使用大写字母，这样就不会有那么多难看的符号了。另外，如果大量使用全局变量，就会使整个程序中到处都是难看的 $ 符号，这也将有助于我们自然地去推广良好的编码风格。

变量名中包含作用域信息的好处是无须再一一寻找变量声明，因为有关变量作用的信息会以一种紧凑的形式展现在你的面前。变量声明用于向编译器提供变量的作用域和类型等信息，与本质的处理没有关系。如果可以的话，我是不想写这种东西的，更不想为了读懂程序而到处去找变量声明，所以才确定了这样的规则，Ruby 也因此没有变量声明之类的东西。变量在最开始赋值时就会被生成，而不再进行变量声明。

慎重起见，这里我再补充一句，我并没有否定声明的优点，特别是类型声明的优点。静态类型语言即使不运行也能在编译时检查出类型不匹配的错误，这让我觉得很了不起。只是我想把精力集中在本质问题上，而且也不想写类型声明，所以目前更倾向于动态类型。

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

给脚本语言增加面向对象功能

在设计 Ruby 时，还有一个从一开始就想好的事情，那就是让这个语言成为真正的面向对象语言。

当时的面向对象语言有 Smalltalk 和 C++，大学研究等领域也在使用 Lisp 系的面向对象语言（Flavors 语言等）。据说还有一门叫作 Eiffel 的语言，主要在国外的金融业等行业中使用，但实际的语言处理器只有商用版本，而且在日本很难获取。

这些原因使得面向对象编程距离我们很遥远，而日常的编程，特别是像脚本的文字处理那种规模又小、复杂程度又不高的编程，一般被认为没有必要使用面向对象编程。

所以当时的脚本语言没有一开始就具备面向对象功能的。即使有支持面向对象编程的功能，也是后来添加上去的，因此大多缺乏一种整体感。

但是，高中时读过的那一点关于 Smalltalk 的资料让我觉得面向对象编程才是理想的编程，我相信在脚本编程领域面向对象也一定是有效的，因此在设计语言时，自然一开始就想朝着面向对象的方向去设计。

单一继承对多重继承

这里让我烦恼的是继承功能的设计。各位读者可能知道，在支持面向对象编程的语言功能中，继承分为单一继承（也叫单重继承）和多重继承。继承是指从现有的类中继承功能，并附加新功能到新的类。其中，作为基础的现有的类（称为父类）的数量只有一个的情况称为单一继承，有多个的情况称为多重继承。

单一继承是多重继承的子集，只要有多重继承就能实现单一继承。多重继承在 Lisp 系的面向对象语言中非常发达，C++ 后来也引入了这项功能，只是不知道在 1993 年的时候这项功能的使用情况如何①。

不过，多重继承有单一继承没有的问题。在单一继承的情况下，类之间的继承关系只是单纯的一列，类阶层整体是树结构（图 1-23）。而多重继承允许多个父类存在，因此类之间的关系呈网状，形成 DAG（Directed Acyclic Graph，有向无环图）结构。在多重继承中，继承的父类也可能同样有多个父类。如果不加注意，类之间的关系马上就会变得复杂。

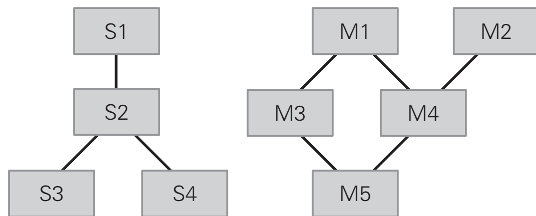


图 1-23　单一继承（左）与多重继承（右）

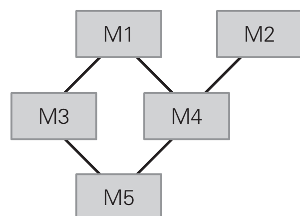
① 根据《C++ 语言的设计与演化》中所说，C++ 是在 1989 年的 2.0 版本中引入多重继承的。1993 年的时候这个功能刚出现不久，可能还没什么人用。

单一继承中，类之间的关系只是单纯的一列，不需要担心继承的优先顺序，搜索方法时也只需按照从下（子类）到上（父类）的顺序查找即可。

但在类关系是 DAG 结构的多重继承的情况下，搜索顺序就不一定是唯一的了（图 1-24）。既有深度优先搜索，也有广度优先搜索，很多支持多重继承的语言（CLOS、Python 等）还采用了这两种方法之外的 C3 搜索方法。

但是，无论选择了哪种方法，都有很难直观地说清楚的情况。这么复杂的继承关系本来就让人难以理解。

那么把多重继承变为简单的单一继承就没有问题了吗？虽然前面说过单一继承的类关系简单，非常容易理解，但这并不代表单一继承就没有问题。



深度优先 ：M5→M3→M1→M4→M1→M2
广度优先 ：M5→M3→M4→M1→M2
C3 ：M5→M4→M3→M2→M1

图 1-24　DAG 的搜索顺序

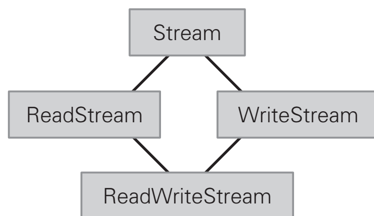
单一继承的问题

单一继承的问题是无法跨越继承范围来共享方法等的类属性。在没有共同的父类的情况下，属性无法共享，只能复制代码。DRY 原则[①] 认为，复制代码是一种恶习，是不好的做法。

我们来看一个实际的例子。Smalltalk 中有管理输入输出的 `Stream` 类，这个类中有负责读取的子类 `ReadStream` 和负责写入的子类 `WriteStream`，还有既可以读又可以写的子类 `ReadWriteStream`。

在支持多重继承的语言中，`ReadWriteStream` 一般会被设计为继承 `ReadStream` 和 `WriteStream` 这两个类（图 1-25a），这是多重继承的比较理想的一个案例。但是 Smalltalk 不支持多重继承，所以就让 `ReadWriteStream` 成为 `WriteStream` 的子类，然后将 `ReadStream` 的代码复制过来（图 1-25b）。假如 `ReadStream` 发生变更，那么复制了 `ReadStream` 代码的 `ReadWriteStream` 也必须相应地进行修改，否则就会出现 bug，这是最糟糕的。

(a) 使用了多重继承的ReadWriteStream



(b) Smalltalk的ReadWriteStream

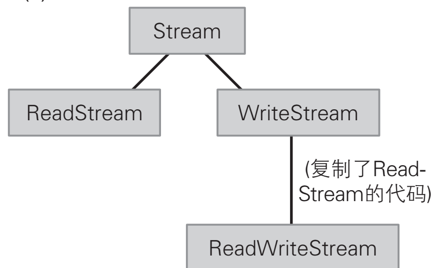


图 1-25　ReadWriteStream

Mix-in

Mix-in 给了我解决这个问题的启示。Mix-in 是在 Lisp 系的面向对象语言 Flavors 中诞生的一项

① DRY 是“Don’t Repeat Yourself”的缩写，这是一个软件设计原则，指软件开发时要避免重复。

技术。Flavors 虽然是支持多重继承的面向对象语言，但是为了减轻刚才讲过的多重继承的问题，该语言对第二个之后的父类进行了如下限制。

- 不得实例化
- 不得有（普通的）父类，可以用其他的 Mix-in

按照这些规则，多重继承的网状结构会变成第一个父类为树结构，第二个之后的父类为像长出了树枝一样的结构。使用这个技术实现和图 1-25 相同的结构，如图 1-26 所示。虽然和直接使用多重继承的结构大不相同，但是保持了单一继承的简洁性，而且不需要复制代码。

Ruby 的模块

Mix-in 的确是个好方法，不过它只是多重继承用法上的一个技巧，并没有强制力，因此我考虑在语言层面上强制使用 Mix-in，也就是准备两种类：一种是用于主继承的普通的类，另一种是只能作为 Mix-in 使用的特殊的类。这个特殊的类需要遵循 Mix-in 的规则，禁止实例化和从普通类继承。

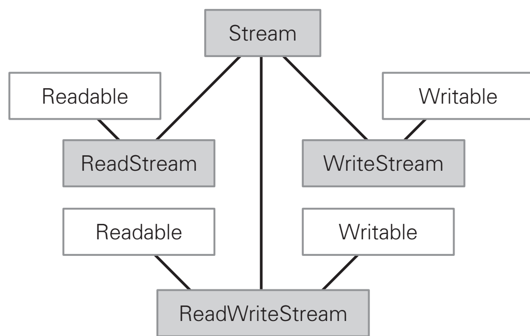


图 1-26　Mix-in

Ruby 的模块就是根据这个想法诞生的。`module` 语句定义的内容恰好满足前面所说的 Mix-in 的性质（图 1-26 的 `Readable` 和 `Writable` 就相当于 `module`）。使用这一技术，我们就能回避多重继承的缺点，降低复杂程度。

差不多和 Ruby 在同一时期，其他语言（例如曾经的太阳微系统公司研究的 Self 语言）也使用 trait 或者 mixin 等名字提供了这样的结构。

错误处理

软件开发中最麻烦的就是错误处理。想打开的文件不存在、网络连接中断、内存不足等，软件不按预期工作的异常情况要多少有多少。像 C 这样的语言在出现了异常的函数调用之后，就需要去检查函数是否正常结束（图 1-27）。

从图 1-27 的代码中可以清楚地看出，“打开文件”的意图只用一行代码就能描述出来，而与此相比，错误处理则非常烦琐。这就违反了“简洁地表达意图”这一 Ruby 的设计原则，无论如何都要想办法解决。

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

```
FILE *f = fopen(path, "r");
if (f == NULL) {    // 文件没有正常打开
  switch (errno) { // 错误的详细信息保存于变量errno中
   case ENOENT:    // 文件不存在
     ...
     break;
   case EACCES:    // 文件访问权限错误
     ...
     break;
    ...
  }
}
```

图 1-27　C 的错误处理

最先考虑的是使用 Icon 语言的错误处理结构。美国亚利桑那大学开发的 Icon 语言中所有函数调用都会返回成功（返回值）或者失败的结果。如果函数调用失败，那么调用者的函数也会运行失败，这一点与 C++ 和 Java 等语言的异常处理结构相似。Icon 的特殊之处在于把失败值当成布尔值处理。

也就是说，下面这行代码因某种原因执行失败时，这个调用者的函数也会执行失败，导致处理中断。

```
line := read()
```

下面的代码表示，当 read() 执行成功时，write() 函数会被执行，否则什么都不做。

```
if line := read() then
   write(line)
```

而下面的代码则表示，只要 read() 执行成功，就会循环去 write() 这个 read() 函数的返回值，read() 或者 write() 中只要有一个执行失败，就会中止循环。

```
while write(read())
```

这种结构不需要使用特殊的语法就可以自然地编写异常处理，这一点很有魅力，但是这样的异常处理不容易引人注意，而且与“普通”语言差别太大，往往让人敬而远之，处理效率方面也令人担心，因此我最终没有采用这种结构。假如我采用了这种结构，那么 Ruby 就和现在大不相同了。

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

如果你想设计一门人气语言，那么在采用不同于其他语言的设计时，就需要考虑它是否会成为语言的亮点。如果你特别执着于这个设计，那么也可以不做出让步，但假如你明明并不是那么在意，却没经过仔细考虑就采用了“怪异的语法”，可能就为今后埋下了祸根。

对异常的关键字有讲究

虽然放弃了使用 Icon 语言式的异常处理，但我还是希望 Ruby 能拥有某种形式的异常处理功能，于是我决定采用 C++ 等语言中的“普通”的异常处理结构（Java 那时还没有出世），不过我对关键字有一点讲究。

C++ 的异常处理用的是 `try~catch` 语句，不过我不怎么喜欢这个关键字。因为 try 给人一种“试试看”的感觉。既然所有的方法调用都可能会发生异常，那么说“试试看”就不太恰当了。另外，catch 这个词也不能让人联想到异常处理。

于是，我从设计代码块时参考过的 Eiffel 语言中借鉴了 rescue 一词。rescue 这个词给人一种“从危险状态中解救出来”的感觉，用在异常处理中正合适。

另外，`ensure` 这个关键字也是从 Eiffel 语言中借鉴过来的，用于无论是否发生异常都执行事后处理（有些语言使用的是 `finally` 这个词）。Eiffel 中的关键字 `ensure` 不是用于异常处理，而是用于表示在 DBC（Design by Contract，契约式设计）中使用的方法运行后应该满足的事后条件。

代码块

最后要说的是代码块。其实原本我并没有特别重视代码块的设计，但后来人们经常说代码块是 Ruby 最大的一个特征，作为 Ruby 的设计者，我也颇感意外。

麻省理工学院开发了一门名为 CLU 的语言，用一句话介绍，CLU 就是面向对象语言的前身或者抽象数据型语言。

CLU 有一些引人注目的特征，其中一个就是迭代器（iterator）。迭代器就是“将循环抽象化的函数”。

迭代器可以通过如下方式调用。

```
for i:int in times(100) do
...
end
```

这段代码调用了名为 `times` 的迭代器函数。迭代器函数是只能在 `for` 语句中被调用的特殊函数。用 CLU 语言实现这个 `times` 函数，如图 1-28 所示。

```
times=iter(last:int) yields(int)
   n:int := 0
   while n < last
     yield(n)
     n := n + 1
   end
 end times
```

图 1-28　用 CLU 语言实现的 times 函数

在迭代器函数中，当 `yield` 被调用时，传递给

yield 的值会被赋值给 for 语句中指定的变量，然后运行从 do 开始的代码块。

同样的处理如果用 C 实现的话，就会用到 for 语句或函数指针。不过使用 for 语句时无法隐藏循环变量以及对内部构造的访问等细节。使用函数指针时，虽然可以隐藏细节（因为 C 语言中没有闭包），但是变量之类的传递会比较麻烦。

而 CLU 的迭代器不存在这样的问题，所以用它来进行循环的抽象化是非常理想的。

反复推敲语法结构

于是我考虑把 CLU 的迭代器引入到 Ruby 里，但是再三思考之后，就觉得直接引入不是很好。CLU 的迭代器确实可以很好地将循环抽象化，不过这种结构也可以用于循环以外的场景中，而 CLU 的语法结构则恰恰阻碍了它在循环以外的场景中使用。

Smalltalk 和 Lisp 中有很多函数和方法可以把函数（Smalltalk 中是代码块）当成参数来传递，然后用于循环等处理。例如 Smalltalk 中使用以下代码就可以将数组的每个元素乘以 2，从而得到一个新数组。

```
[1,2,3] collect: [:a| a * 2].
```

如果这个处理用 CLU 的语法来写，就可能会写成下面这种形式，看起来不太直观（这里只是模拟了 CLU 的写法，实际上 CLU 是不能这么写的）。

```
for a in [1,2,3].collect() do
  a * 2
end
```

难道就没有更好一点的写法吗？当时我的大女儿刚出生不久，晚上总是不睡，我就一边哄她睡觉一边琢磨这条语句的写法。

一开始想到的语法是下面这样的。

```
do [1,2,3].collect using a
  a * 2
end
```

显然这个语法受到了 CLU 的影响。using 这个关键字是从一个叫 Actor 的 PC 语言中借鉴过来的，这里所说的 Actor 与并发编程的 Actor 模型没有任何关系。然而即使这样写，也没能达到像 Smalltalk 的代码块和 Lisp 的匿名函数那样易于理解的程度。

经过反复推敲，最后实现的语法如下所示。

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

```
[1,2,3].collect {a| a * 2}
```

这个语法已经非常接近 Smalltalk 了。只有一个表示变量的“|”也是受了 Smalltalk 的影响。之后该语法又被不断完善：没有变量时用两个“|”表示省略；在与其他以 end 结尾的语句混合使用时，为保持风格一致，使用 do~end 表示代码块。

扩大了使用范围

原本为了循环的抽象化而参考 CLU 语言设计的代码块，在被引入 Ruby 后出现了各种各样的用法，比如下面这些。

- 循环的抽象化（必不可少）
- 指定条件（select 等）
- 指定回调代码（GUI 等）
- 线程及 fork 的运行部分
- 指定作用域（DSL 等）

代码块的使用几乎遍及各个领域，真是让人惊讶。

然而代码块只不过是在 Lisp、Smalltalk 以及其他函数式语言中广泛使用的高阶函数（把函数作为参数使用的函数）的特殊语句而已，所以自然能实现以上这些用法。但由于它是专门为循环的抽象化而设计的，所以会有一些限制，而让我意外的是，这些限制完全没有妨碍到它的使用。

刚才提到的限制指的是：

- 没有直接编写函数对象（function object）的语句（直到版本 1.9 中有了 lambda 表达式）
- 由于带代码块的调用被整合在了语法中，所以一个方法只能有一个代码块

实际上正是有了这些限制，代码在大多数情况下才变得更容易理解、更容易设计了。真是塞翁失马，焉知非福。

根据某项调查显示，在函数式语言之一的 OCaml 的标准库里大量存在的高阶函数中，有 98% 都只有一个函数参数。Ruby 的方法参数中只能有一个代码块的限制并没有带来什么问题，或许也是出于同样的原因吧。

语言设计的秘诀

看到这里，相信各位读者也应该知道一些语言设计的秘诀了。

第一，要充分调查现有语言存在什么样的问题，以及有哪些解决办法。积累这些琐碎问题的解决经验，有助于完成一个良好的设计。

第二，对个性的追求要限定在“某一点”上。我在介绍 Icon 的异常处理时也提到过，独具一格的语法可能会给初学者带来挫败感，在“某一点”以外的地方采取保守的态度也会给语言带来人气。但是过于保守会使语言在技术层面上平淡无趣，吸引不到用户，所以很难掌握好这个平衡。

第三，站在客观的角度。我们都知道，如果大家一起商量如何设计，最终并不会产生什么好的结果。因为大家会相互妥协以达成统一意见，从而舍弃设计中独特的部分，也就失去了这种独特设计的优点。因此，就算跟人商量，也要仅限于寻求意见。如果最终责任不是由一个人来担负，就无法做出好的设计。

我在设计代码块时，总是跟婴儿和泰迪熊商量，这也是一个办法。也许你会觉得这么做有点傻，但这么做的话，即使对方没有反应，通过说明自己的想法也有助于进一步深入思考，有时甚至会想出更好的主意。

小结

在 1-4 节和 1-5 节，我向大家介绍了 Ruby 设计早期的一些想法，大家感觉怎么样？

即使是一个细微的语言细节，设计者也要在经过各种研究和思考之后才能决定。不知大家是否感受到了这一点。

不仅是语言，所有的设计都是一个权衡折中的过程。完美无缺的设计是不存在的，存在的只是在某种条件下更好的选择。能否让这个选择适用的范围更广并尽可能地接近完美，就要看设计者的能力了。

不过设计语言是一个漫长的过程。即使是 Ruby 这门在大家的认知中还算比较新的语言，从开始开发也已经经过二十多年了。这期间计算机的性能不断提高，环境也发生了变化，于是 Ruby 又出现了新的需要权衡折中的地方。比如 Ruby 诞生之时多核计算机还没有普及，所以不会要求线程去有效利用多核 CPU，然而现在面向个人的计算机也都已经是双核、四核的了。

为了应对这种环境的变化，我每天都在重新思考语言的设计和实现。

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

时光机专栏

语言设计的秘诀适用于所有设计

本节是 2014 年 8 月刊中刊登的内容。我接着上一节的内容讲述了一些语言设计背后的事情，介绍了 Ruby 的变量命名规则、面向对象功能的设计、异常处理等的设计背景。

一般来说，在介绍一门语言时，大家往往倾向于介绍该语言最终的样子，而不去讲解为什么是这样的。这次我作为 Ruby 的开发者深入讲述了平常不会触及的一些语言设计的相关内容（我还算是讲得较多的）。我记得自己在写稿子的时候也非常开心。

我在写本节的时候还没有明确意识到，“デザイン”和“設計”这两个词对应的英文都是“design”[①]，但是在日语中，总感觉“デザイン”和“設計”意思不太一样。

日语的“デザイン”给人的感觉是“决定样式”，而“設計”则是“考虑结构”。我最擅长的就是决定编程语言具有什么样的功能和语法，这也是我的工作。我把心思都放在了语言的“美观”上，所以最近开始使用“言語デザイン”（language design）这个词了。大家是否也觉得“言語デザインナー”（language designer）这个头衔很酷呢?

在本节后半部分介绍的语言设计的秘诀中，我讲到了非常重要的内容（算是自卖自夸吧）。这些原则不仅在语言设计上，而且在所有的软件设计上都是通用的。虽然我在编程以外的领域缺乏经验，但我认为这些原则已经超越了编程，适用于设计和需求的确定等所有需要决策的领域。

① “design”这个词在日语中有两种表示方法，一种是根据“design”这个单词的发音产生的外来语“デザイン”，另一种是“設計”。——译者注

第 2 章

新语言Streem的设计与实现

2-1 抽象的并发编程

多核时代已经到来，即使是日常的编程场景也开始需要进行并行编程了。从本节开始，我们将研究新时代的并行编程，设计支持并行编程的新语言。不过在这之前，我们先来探讨一下为什么需要支持并行编程的语言。

如今，电器店里卖的普通的计算机都已经搭载多核 CPU 了，就连智能手机也都是 4 核或者 8 核的。多核如此普及是有一定原因的。

在过去的 50 年间，以 CPU 为首的半导体的集成度按照摩尔定律实现了指数级增长，CPU 的性能也随之提高，但这种情况在前不久出现了终结的苗头，因为 CPU 的性能已经没有提升空间了。想必各位读者也注意到了，近来 CPU 频率一直维持在 2 GHz 上下，不再像以前那样大幅度提升了。

多核化接过大旗

近几年，由于大规模集成电路（LSI）的集成度过高，电路变得只有几个原子并起来那么宽，所以出现了电子穿过绝缘层的“量子隧穿效应”等量子力学方面的问题。另外，电路的密集也导致了热密度上升。最近，CPU 内核的热密度已经超过了电热板。如果没有好的冷却办法，过不了多久，一插上电源电路就会被烧掉。特别是进行复杂处理的 CPU 内核，不管是电路密集度还是热密度都已经接近极限，事实上已经很难指望使用单一内核来实现性能的大幅提升了。

但并不是所有电路的热密度和复杂度都这么高。构成内存和总线的电路就比 CPU 内核的简单一些，不容易出现热密度的问题。因此可以将多个内核放在一个芯片上来降低平均热密度，这也是多核化的一个目的。

这种硬件进化的趋势在未来一段时间内都不太可能被颠覆。目前单个 CPU 内核很难实现性能的大幅提升，不管你是否愿意，想要最大限度地发挥新计算机的性能，就只能依赖多核。

并行与并发编程

要想有效利用多核，就需要同时进行多个处理。在这种同时进行多个处理的编程中，需要注意“并行”和“并发”两个术语。

并行的英语是“parallel”，意思是同时进行多个处理，并发的英语是“concurrent”，意思是至少看上去是在同时进行多个处理。大家明白其中的区别了吗？

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

并发编程是指，即使只有一个 CPU，也要把多个处理分割为小的任务交替执行，让这些处理看上去像是在被同时执行一样。但实际上在 CPU 只有一个的情况下，一次只能执行一个处理，所以无法实现并行编程。由于多核环境中有多个 CPU，所以如果可以把处理适当分配给多个内核，就可以实现并行编程。

对大部分软件开发者来说，并发编程比较重要，因为现在大部分操作系统和运行环境会根据 CPU 的数量切换到相应的运行模式：如果只有一个 CPU，就切换到看上去在同时执行处理的运行模式；如果有多个 CPU，则切换到真正在同时执行处理的运行模式。也就是说，只要操作系统和运行环境的开发者注意并行和并发的区别即可。

接下来我们来了解一下迄今为止都有哪些支持并发编程的结构。

在支持并发编程的结构中，最具代表性的是进程和线程。现在大部分操作系统提供了这两种结构。

■ 进程

在支持并发编程的结构中，最早的就是进程。比进程历史更悠久的还有任务（task），不过任务在功能上与进程并没有太大的差别，而且也没有在 Linux 编程中出现，所以本书中就略去不谈了。

进程是操作系统中表示“运行中的程序”的结构。现在的操作系统[①]可同时运行多个程序。

UNIX 的 fork 可以进行复制

UNIX 中使用 `fork` 系统调用创建新的进程。`fork` 系统调用会创建运行中的程序的副本（又一个进程）。

使用 `fork` 进行复制后，`fork` 会在被复制的程序（被复制的进程，也叫父进程）中返回新的进程的 ID（整数），在由复制产生的新的进程（子进程）中返回 0。由于子进程复制了父进程，所以之前绝大部分的运行结果，比如变量值、内存分配等是父进程的副本。但由于 `fork` 的返回值不同，所以接下来父进程和子进程会各自进行处理。

实际上在大部分情况下，子进程中（在根据需要稍微进行一下准备工作之后）会启动别的程序。在自己的进程中启动程序就需要使用 `exec` 系列的系统调用。`exec` 系列的系统调用会在同一个进程中替换要运行的程序。

C 语言的系统调用有些复杂，这里我们用 Ruby 程序进行展示（图 2-1）。

---

① 以前的操作系统，比如 MS-DOS 只能同时运行一个进程。当时，支持多个程序同时运行的操作系统称为“多任务操作系统”。不过最近这样的操作系统已经极其普遍了，“多任务操作系统”这个词也就完全没人使用了。

```
pid = fork()  # 在使用Ruby的情况下，子进程中会返回nil
if pid
  # 检查子进程是否已运行完毕
  Process.waitpid(pid)
else
  # 启动echo
  exec "echo", "hello world"
end
```

图 2-1 使用 Ruby 进行 fork 和 exec

fork 用来复制，exec 用来替换进程中运行的程序，这种 fork 和 exec 组合使用的形式是类 UNIX 操作系统所特有的。而其他很多操作系统（Windows 等）则会提供用于直接启动程序的 spawn 系统调用。

创建进程的开销很大

对于通过这种方式启动的多个进程，操作系统会适当分配运行时间，让它们至少在表面上看起来是在同时运行的。如果计算机有多个 CPU（而且操作系统也支持多个 CPU 的话），操作系统就能把进程分配给多个 CPU，让它们并行运行。

进程的特征是各个内存空间都是相互独立的。fork 产生的进程复制了原进程，因此在子进程中修改内存状态不会影响到父进程。即使在子进程中做一些出格的操作，父进程也是安全的。

进程的缺点是开销大。由于复制的是整个内存空间，所以用 fork 创建进程的开销非常大。最近很多操作系统采用了 CoW（Copy on Write，写时复制）等策略，CoW 可以使父子进程共享内存空间，在需要修改时再进行复制。但即便如此，复制的开销也无法忽视。

比如以前被广泛使用的动态 Web 页面技术 CGI[①] 渐渐失去了人气，其最大原因就是进程的创建开销大。

进程间通信困难

进程的另一个缺点就是进程间通信困难。内存空间相互独立，从安全性的角度来看的确是一个优点，但同时也阻碍了多个进程之间的信息交换。

在类 UNIX 操作系统中，进程之间信息交换的手段有限，主要有父子进程通过共享管道的方式使用管道数据流通信、使用套接字通信、通过文件来交换信息、共享内存等方式。另外还可以通过

① CGI（Common Gateway Interface，通用网关接口）是一种动态 Web 页面技术，针对 Web 页面的请求启动一个进程，并将这个进程的输出传送给浏览器，以此来提供动态 Web 页面。

信号量（semaphore）和信号（signal）等机制进行进程之间的并发控制。

不管是哪种共享信息的手段，最终传递的都是字节序列。要想发送数值、数组和映射（map）等字符串以外的数据，就需要先将它们转换成字符串发送，然后解析字符串并将其恢复为数据。通信数据量增多时，这种转换开销也不容忽视。

■ 线程

线程是在一个程序之中共享内存空间，同时对多个处理流程进行控制的结构。由于不需要复制内存空间，所以线程要比进程的创建开销小，这是它的一个很明显的优点。

在我刚进入编程领域时，线程还不是一个随处可用的功能，但如今却大不相同。线程不仅在 POSIX 中被标准化，还可以在 Windows 系统（不过 API 有所不同）中使用。

通信开销小

得益于内存空间的共享，线程间的通信开销很小，这是线程的主要特征。不管是字符串、整数还是结构体，无论多么复杂的数据结构都可以实现零开销访问。

邮
电

但是共享内存空间带来的也不全是好事。多个处理同时执行就意味着不知道什么时候数据就会发生改变，可能还没等你反应过来，数据就已经乱套了。

另外，数据的一致性也无法得到保证。比如图 2-2 的程序，从 a 中减去 1000，又在另一个线程中给 a 加上 1000，最终结果应该和原来的值 5000 一样，但有时候返回的却是其他的值。在“a = a + 1000”部分，也就是从取出 a 的值再到把新值赋给 a 之间的这个微妙的时间点上，如果其他线程正好修改了 a 的值，就会得到和预期不同的结果（图 2-3）。

```
a = 5000
th = Thread.fork{
  a = a + 1000
}
a = a - 1000
th.join
puts "a=",a    # 结果是?
```

图 2-2 有问题的线程程序

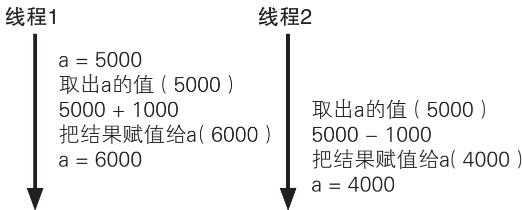


图 2-3 线程导致数据一致性被破坏

并发控制变得复杂

要避免这个问题，就需要使用“并发控制”，避免同时访问必须保持数据一致性的地方（图 2-4）。

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

在修改可能会被多个线程访问的数据（本例中的变量 a）之前，可以通过加锁的方式来保证只有一个线程能访问这个数据。

前面也提到过，线程中会有多个处理流程同时运行，所以即使多次执行同一个程序，问题也是时有时无。像这种可重复性较差的 bug 俗称“海森堡 bug”[①]。这种 bug 很难被发现，处理起来比较棘手。

线程的缺点和难点并不止这些。的确，由于不需要复制内存空间，线程比进程的创建开销要小，但当创建线程时仍然需要进行系统调用。

当进行系统调用时，不仅需要切换到能执行特权指令的内核空间，还需要耗费大量的 CPU 指令。如果创建线程的频率很高，那么这个开销也是不能忽视的。而且每次生成新线程时都需要为线程栈空间分配几兆的内存。如果要创建 1000 个线程的话，光是这些线程就需要耗费 1 GB 的内存。

```
m = Mutex.new
a = 5000
th = Thread.fork{
  m.lock
  a = a + 1000
  m.unlock
}
m.lock
a = a - 1000
m.unlock
th.join
puts "a=",a    # 结果是5000
```

图 2-4 通过并发控制保持数据的一致性

■ 理想的并发编程

从开销的角度考虑，操作系统提供的用于并发编程的进程和线程并不理想，而且很多时候还需要注意并发控制等。

于是我不断摸索，找到了比进程和线程更高级的并发编程结构。下面就对其中一部分进行介绍。

Actor 模型

Actor 模型是美国麻省理工学院的卡尔·休伊特（Carl Hewitt）于 1973 年左右设计的一个计算模型。在 Actor 模型中，所有对象都有 Actor 这个独立的处理流程。

Actor 之间可以传递异步消息。异步指的是消息发送之后不需要等待处理结果，通过接收从对方 Actor 返回的消息来获取消息的应答。

面向对象本来就是为操作用于模拟的对象而诞生的，因此可以说 Actor 模型推动了它的发展。

虽然出现了一些可以用 Actor 表现所有数据的编程语言，但这些语言都没能普及开来。比如东京大学开发的 ABCL/1 等，并未得到实际应用。

① “海森堡 bug”这个名字取自于提出了量子力学“不确定性原理”（俗称“测不准原理”）的物理学家海森堡。

Erlang 的“进程”

直接实现 Actor 模型的语言虽然没能得到普及，但受 Actor 模型影响的语言却有不少，其中具有代表性的是 Erlang。

诞生于 1986 年的 Erlang 是支持分布式并发编程的高可靠性语言。据说它是瑞典爱立信公司为了开发电话交换机等软件而开发的语言。

虽然 Erlang 是函数式语言，没有对象，但是有“进程”这个概念。据 Erlang 的设计者乔・阿姆斯特朗（Joe Armstrong）所说，之所以称为“进程”，是因为它不共享内存，与线程不同。但老实说，这个概念非常容易与操作系统的进程混淆，真希望他能改一改这个叫法。

Erlang 的进程与操作系统的进程相比非常轻量，平均每个进程只消费几百个字节。不仅如此，创建是在用户模式下进行的，没有系统调用，因此花费的时间也很少。

这个进程相当于 Actor 模型中的 Actor。Erlang 语言处理器会根据计算机的 CPU 内核数创建线程，然后将各个 Erlang 进程分配给这些线程去执行，因此在多核环境中可以最大程度地使用多个 CPU 内核。

容易共享数据

我们可以向 Erlang 的进程发送消息。消息发送是单方向异步的，也就是说不需要等待结果返回。如果想要得到结果，就需要在消息中添加发送消息一方的进程 ID，这样对方进程就会把处理结果作为消息返回来（图 2-5）。

Erlang 是函数式编程语言，几乎所有的数据结构都是不可变的（immutable）。Erlang 进程之间除了通过消息共享数据以外，基本上没有其他数据共享的方法（实际上还有内嵌数据库这种信息交换方式）。因此，在介绍线程时提到的问题基本上不会在 Erlang 中发生。

这种发送消息的程序经常出现一个问题，就是程序 bug 使消息发送失败，进而使整个程序停止运行。不过高可靠性的 Erlang 很善于编写错误处理代码，比如可以轻松编写出消息接收超时这种情况的处理代码，也很容易编写出检测到进程异常退出后重启进程的错误处理代码。

```
-module(pingpong).
-export([start/0, ping/2, pong/0]).

% 此处代码只有在N为0时被调用
% 向Pong发送finished消息
ping(0, Pong_PID) ->
    Pong_PID ! finished,
    io:format("ping finished~n", []);

% 此处代码只有在N为0时被调用
% 用"Pong_PID ! {ping, self()}"发送消息
```

```
% 用receive接收消息
% 接收pong消息，然后循环
% 由于Erlang没有正式的循环语法，所以使用递归
ping(N, Pong_PID) ->
    Pong_PID ! {ping, self()},
    receive
        pong ->
            io:format("Ping received pong~n", [])
    end,
    ping(N - 1, Pong_PID).

% Pong的实现
% 收到finished消息时结束程序
% 收到ping消息时返回pong，循环这一处理
pong() ->
    receive
        finished ->
            io:format("Pong finished~n", []);
        {ping, Ping_PID} ->
            io:format("Pong received ping~n", []),
            Ping_PID ! pong,
            pong()
    end.

% 程序从这里开始
% 用spawn创建两个进程，让它们进行pingpong
start() ->
    Pong_PID = spawn(pingpong, pong, []),
    spawn(pingpong, ping, [3, Pong_PID]).
```

图 2-5 Erlang 的消息发送

Go 的 goroutine

谷歌公司开发的 Go 语言与 Erlang 相似，也支持以消息通信为基础的并发编程。Go 语言中用 goroutine 这个词代替进程这一术语，在名称上不会引起混乱，这一点非常好。goroutine 和 Erlang 的进程一样，内存消费量少，创建的时间成本小，而且可以在一个进程中大量创建。另外，在根据 CPU 数量将处理分配到多个线程执行，从而有效使用多核这一点上，goroutine 也和 Erlang 相同。

Go 语言使用 go 语句创建新的 goroutine。另外，Erlang 中进程自身是消息发送的目的地，而在

Go 语言中则要发送消息到 chan（channel，通道）对象。因此，Erlang 中的如下处理

- 用 spawn 创建进程
- 向进程发送消息
- 用 receive 接收消息

在 Go 语言里会变为

- 事先创建 chan
- 创建 goroutine，向它传递 chan 对象
- 向 chan 发送消息
- 用 select 从 chan 接收消息

Go 语言有两个特点，一个是必须显式地传递消息通信的通道，另一个是无法得到 goroutine 的 ID。图 2-6 是用 Go 语言重写的 Erlang 的 pingpong 程序。我对 Go 语言还不是很熟悉，所以图 2-6 的代码可能有不符合 Go 语言编码习惯的地方。

```
package main
import "fmt"

func ping(n int, ping, pong chan int) {
  for {
    pong <- n;
    if n == 0 {
      fmt.Println("ping finished");
      return;
    }
    <- ping
    n = n - 1;
  }
}

func pong(ping, pong, quit chan int) {
  for {
    n := <- pong
    if n == 0 {
      fmt.Println("pong finished");
      quit <- 0;
      return;
    }
```

```
    fmt.Println("pong sending ", n);
    ping <- n;
  }
}

func main() {
    pingc := make(chan int);
    pongc := make(chan int);
    quitc := make(chan int);
    go pong(pingc, pongc, quitc);
    go ping(3, pingc, pongc);
    <- quitc;
}
```

图 2-6 用 Go 语言编写的 pingpong 程序

老实说 Erlang 的代码更为简洁，Go 语言中必须显式地创建并传递通道这一点有些麻烦。但 Go 是静态类型语言，因此我猜想开发者在设计 Go 时希望它能够更加灵活，比如需要显式地声明作为消息传递的是哪种类型的数据、区分使用多个通道等。

Clojure 的 STM

在介绍线程时，我们提到过在存在信息共享的情况下如果不进行适当的并发控制，数据就会遭到破坏（丧失一致性）。

类似的问题也会在接受多个客户端访问的数据库中发生。数据库在设计上需要遵循 ACID 原则，ACID 取自于 Atomicity（原子性）、Consistency（一致性）、Isolation（隔离性）和 Durability（持久性）这 4 个要素的首字母。

原子性是指对数据库的操作要么全部执行，要么就什么都不做，不允许在中间某个环节结束。因为无法再进行分割，所以用了“原子”这个词。

一致性是指保证数据库的状态永远符合预设的规则，不符合预设规则的处理会被取消。例如在存款账户管理系统中，如果设定余额必须永远为正数，那么取出超过余额的钱的操作就不满足预设规则，无法进行。

隔离性是指无法从外部看到保持了原子性的一系列处理的中间状态。

持久性是指在保持了原子性的一系列处理结束时，其结果会被保存起来，不会丢失。

引入数据库的概念

在数据库中，我们把原子性的单位称为事务（transaction）。事务中可以查看和更新数据，但是

外部看不到处理的中间状态，只有在事务成功之后外部才能看到更新的结果。如果事务处理出于某些原因而失败，那么之前的所有更新都会被取消。

把数据库中使用的事务概念引入普通编程中去的是 Clojure 中的 STM（Software Transactional Memory，软件事务内存）[①]。

图 2-7 是在 Clojure 中使用 STM 的程序示例。事务在 `dosync` 包围的部分中用 `ref` 生成共享信息，用 `deref` 查看信息，用 `ref-set` 更新信息。Clojure 的数据结构基本上是不能修改的，因此在原则上就需要使用事务去更新信息。

```
(define a (ref 5000))
(define th #(Thread. (fn []
  (dosync
    (ref-set a (+ (deref a) 1000))))))
(dosync
   (ref-set a (- (deref a) 1000)))
(.join th)
(dosync
   (println (str "a=" (deref a))))
```

图 2-7　Clojure 的 STM

小结

本节介绍了并发编程的必要性以及各种语言为支持并发编程所引入的结构。在 2-2 节，我们将在本节内容的基础上探讨什么是理想的并发编程语言。

时光机专栏

多核时代需要的编程语言

本节是 2014 年 12 月刊中刊登的内容。虽然本节只介绍了并发编程的背景，但我们也算进入到了支持并发编程的语言的设计环节。

我在很早之前就想设计一门支持并发编程的语言。原本是想让 Ruby 支持并发编程，所以在很早的时候就引入了线程。但是在开发 Ruby 的 20 世纪 90 年代，计算机还都是单核的，不需要考虑并行计算的运行环境，于是我就没有考虑多核环境的应对策略。这也导致在多核计算机普及的近几年，屡屡能听到对 Ruby 的线程实现不满的声音。

不仅如此，一想到要使用线程，就会让人感受到前文所说的那种开发难度。近几年，多核环境被充分使用，人们对更简单、更准确的并发编程语言产生了更大的需求。想必本节介绍的 Erlang 和 Go 也是这种需求催生的产物。不过，我认为还可以实现其他抽象度更高的并发编程，这也正是我开发 Streem 语言的原因。

① 需要注意的是，STM 的结果只是保存在临时存储数据的内存中，不能充分满足持久性，这一点与数据库不同。

2-2 新语言Streem

我们在2-1节介绍了并发编程的基础知识。本节将考察在多核已经普及的21世纪，并发语言应该是什么样的，然后再向大家介绍按照这种要求设计的新语言Streem。

在多核环境普及的现在，人们开始重新审视 shell 脚本的价值。shell 脚本的基本计算模型是用管道把多个进程连接起来。在支持多核的操作系统环境下，这些进程会被分散到多个内核，从而自动形成有效使用多核的形式。只要恰当地选择计算模型，就能以自然的形式进行并发运行。图 2-8 就是一个很好的例子。

据说有些业务系统的核心部分也是用 shell 脚本处理的。这些系统使用 shell 脚本对信息进行筛选和加工，与以前的做法相比，变更成本更低，灵活性也更高。

●管道

command1 | command2

●用两个CPU运行管道处理

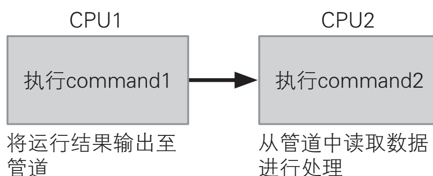


图 2-8　并发 shell 脚本

现在的 shell 脚本还不够理想

但 shell 脚本也有不够理想的一面。

首先，前面也提到过，操作系统进程的创建开销非常大，而 shell 脚本以极小的粒度大量创建进程，这会对性能造成不利影响。

再来说一下开销。由于连接进程的管道只能发送字节序列，所以在传递有结构的数据时，发送者需要将数据转换为字节序列，而接收者则需要再将其转换回数据。比如用逗号作分隔符的 CSV（Comma Separated Values）以及用于 JavaScript 对象表示的 JSON（JavaScript Object Notation）就较为常用。把数据转换为这种形式的字节序列，再解析字节序列转换回数据，这其中的开销不容小觑。

大多数人使用多核是因为想要处理大量的数据，或者拥有更好的性能。可想而知，数据转换和进程创建的开销是巨大的。这也是 shell 脚本的一个缺点。

另外，构成管道的进程的命令是零零散散地开发出来的，这些命令在用法上各不相同，难以熟练掌握。

21 世纪的脚本语言

这么说来，如果能够把 shell 脚本的优点和通用编程语言的优点结合起来，就能产生最强语言了。

那么我们就来思考一下最强语言应该具备什么样的条件。

第一个条件是轻量级并发运行。操作系统级别的进程和线程的创建开销较大，所以尽量不去创建它们。可行的实现方法是，在一个操作系统进程之中预先创建与计算机的内核数（+α）相等的线程，然后让它们交替运行。被公认为是并发语言的 Erlang 和 Go 也采用了这种实现方式（图 2-9）。Erlang 的进程和 Go 的 goroutine 在这里被称为“任务”。

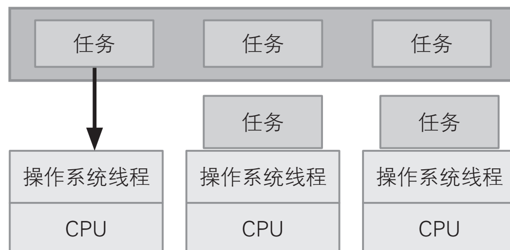


图 2-9　并发结构的架构

轮到的任务在操作系统线程中并发运行。在 I/O 等待等情况下进行替换

第二个条件是要排除并发运行中的竞争条件，具体来说就是排除“状态”。变量和属性的值发生变化时会产生不同的状态，如果运行时机不对，就可能会出现问题，因此就让所有的数据都不可变，以此来避免这类问题的发生。

第三个条件是计算模型。虽然 2-1 节介绍的线程那样的模型表达力很强，但是太过自由，写出来的代码不容易理解。于是我参考了 shell 的运行模型，引入了抽象度较高的并发计算模型。虽然抽象度高了，但是表达的自由度变低了，所以还需要花心思去编写，不过最终的代码应该是易于调试的。

■ 新语言 Streem

下面我们就来设计满足这些条件的语言。因为是以流为计算模型的语言，所以就命名为“Streem”。虽然“流”的英文拼写是 Stream，但是这个词太常用，在搜索引擎上的可搜索性太差，而 Stream 的仿词 Streem 除了拼写错误的情况以外基本没有人使用，而且幸运的是，streem.org 的域名也还能注册。

我的个人信条是语言设计先从语法开始。不过 Streem 是以 shell 为基础的，所以语法上并没有什么特别之处，基本语法如下所示。

```
表达式 1 | 表达式 2 | ...
```

`cat` 命令用于从标准输入读取数据并打印到标准输出。使用上述语法编写与 `cat` 命令作用同样的程序，如下所示。

```
stdin | stdout
```

非常简单。stdin 和 stdout 是表示标准输入输出的常量对象。在 Streem 程序中，stdin 将读取了标准输入的行（字符串）连续不断地传递下去，看起来就像是在流动一般，我们把这种表现数据流动的对象称为“流”。stdout 则是把接收到的字符串流转到外部世界（标准输出）的流。

指定文件名从文件读取数据的代码如下所示。

```
fread(path)
```

指定文件名将数据写入文件的代码如下所示。

```
fwrite(path)
```

这两个函数会分别返回读取用和写入用的流。

表达式

表达式有常量、变量、函数调用、数组表达式、map 表达式、函数表达式、运算表达式和 if 表达式，各表达式的语法如表 2-1 所示。对于有一般语言的编程经验的读者来说，这些表达式应该不是很难。

表 2-1　Streem 的表达式

名称	语法	示例
字符串常量	" 字符串 "	"foobar"
数值常量	数值	123
变量	标识符	FooBar
函数调用	标识符 (参数 ...)	square(2)
方法调用	表达式 . 标识符 (参数 ...)	ary.push(2)
数组表达式	[表达式 , ...]	[1,2,3]
map 表达式	[表达式 : 表达式 , ...]	["foo":1,"bar":2]
函数表达式	{ 变量 ...-> 语句 ...}	{x->x+1}
运算表达式	表达式 运算符 表达式	1+1
if 表达式	if (表达式) { 语句 ...} else { 语句 ...}	if (true) {0} else {2}

赋值

Streem 的赋值有两种写法。一种与一般语言相同，使用等号赋值。

```
ONE = 1
```

另一种是使用“=>”进行反向赋值。上面用等号赋值的代码如果用“=>”重写，则如下所示。

```
1 => ONE
```

在把管道的运行结果保存到变量时，第二种写法延续了流的处理，非常便捷。

无论是哪种形式的赋值，为了避免状态变化，都需要遵守下列规则。

- 规则 1：不能对同一个变量多次赋值，在作用域中只能对一个变量赋值一次
- 规则 2：仅在交互执行环境的顶层作用域（top level）下允许对一个变量进行重复赋值，这可以看作是对变量名相同的不同变量进行的赋值

多条语句

Streem 中可以有多条语句前后排列，语句与语句之间通过分号“;”或换行进行分隔。可以认为有多条语句时会按照先后顺序执行。如果它们之间没有依赖关系，那么在实际执行时就可能会被并行执行。

Streem 程序的例子

接下来我们看几个 Streem 程序的例子。

刚才我们实现了 cat，现在来看一个稍微有些不同的例子，使用 Streem 编写经常作为例题出现的 FizzBuzz 游戏（图 2-10）。这个游戏的规则是：参加者从 1 开始报数，数字能被 3 整除时回答“Fizz”，能被 5 整除时回答“Buzz”，既能被 3 也能被 5 整除时回答“FizzBuzz”。

使用 seq 函数可以创建从 1 开始到指定数字为止的数列。把指定函数传给管道，函数就会作用于数列中的各个元素，其操作结果会传递给下一个管道。stdout 把接收到的数字（的数列）打印出来。

```
seq(100) | map{x->
  if (x%15==0)
    "FizzBuzz"
  else if (x%3==0)
    "Fizz"
  else if (x%5==0)
    "Buzz"
  else
    x
} | stdout
```

图 2-10 用 Streem 编写 FizzBuzz

这么一看，大家是否觉得用 Streem 编写的管道代码非常直接明了呢?

```
1 对 1、1 对 n、n 对 m
```

Streem 可以轻易地编写出图 2-10 那样的对值的序列进行加工之后再输出的程序，但程序也不都是这种一对一的关系，也有像 grep（单词搜索）这种搜索符合条件的文字的类型，还有像 wc（单词计数）这种进行统计的类型。

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

Streem 中有几个关键字会在这些场景中使用。

emit 用于运行一次返回多个值的场景。传给 emit 多个值就会返回多个值，所以下面两行代码表达的是同一个含义。

```
emit 1, 2
```

```
emit 1; emit 2
```

另外，在数组前加上“*”时，emit 会一次返回所有数组元素，也就是说，

```
a = [1,2,3]; emit *a
```

等同于以下代码。

```
emit 1; emit 2; emit 3
```

图 2-11 是一段使用 emit 的示例代码，这个程序会把从 1 到 100 的数字各输出两次。

```
# 循环输出每个值两次
seq(100) | map{x->emit x, x} | stdout
```

图 2-11　emit 示例代码

return 会结束函数运行并返回值。return 可以返回多个值，在这种情况下会对多个值进行 emit。另外，在函数正文只有一个表达式的情况下，即使没有 return，这个表达式的值也会成为返回值。

使用 emit 和 return 可以生成比接收到的值更多的值。反之，使用 skip 则可以生成比接收到的值更少的值。skip 会结束本次循环，不会输出值。图 2-12 是一段使用了 skip 的示例代码，这段代码会从 1 到 100 的数字中过滤掉偶数。

```
# 选择奇数
seq(100) | map{x -> if (x % 2 == 0) {skip}; x} | stdout
```

图 2-12　skip 的示例代码

不可变

前面已经介绍过，为了避免竞争，Streem 中所有的数据结构都是不可变的，数组与映射（相当于 Ruby 的散列）也是不可变的。增加元素时，不是去改变现有的数据，而是在原有数据的基础上增加元素，产生新的数据（图 2-13）。

```
a = [1,2,3,4]    # a是包含4个元素的数组
b = a.push(5)    # b是在a的末尾增加5的数组
                 # a没有变化
```

图 2-13　不可变数据的更新

普通的面向对象编程都是允许属性（实例变量等）变化的，而 Streem 不允许这么做，使用时需要注意。这一点可以说与函数式编程语言很像。

单词计数

接下来我们试着用 Streem 写一下经常在 MapReduce 计算模型的例子中出现的单词计数程序（图 2-14）。

```
stdin | flatmap{s->
  s.split(" ")
} | map{x->[x, 1]} | reduce_by_key{
  k,x,y -> x+y
} | stdout
```

图 2-14　使用 Streem 编写的单词计数程序示例

首先来介绍一下图 2-14 的程序中首次出现的语法。调用 `flatmap` 函数的地方有一段像 Ruby 代码块一样的代码。Streem 有一种语法糖——当函数附加在参数列表之后时，函数会作为最后的参数追加到参数列表中。也就是说，下面第一段代码是第二段代码的另一种形式。

```
flatmap{s->s.split(" ")}
```

```
flatmap({s->s.split(" ")})
```

这么做是为了让普通的函数调用结构可以像 Ruby 代码块一样美观。

我们来看看图 2-14 的代码做了什么。首先从 stdin 逐行接收数据，通过 split 将数据分割为单词，然后通过 flatmap 把单词展开为流。之后通过 map 函数转换为 [单词 , 1] 这样的数组形式，通过 reduce_by_key 创建单词与单词出现次数的映射。reduce_by_key 接收 [键 , 值] 这样的 2 个元素的数组流，然后返回按各个键分组之后的流。当已经出现过的键再次出现在流中时，作为参数传递的函数会通过 3 个参数（键 , 旧值 , 新值）被调用，而且函数的返回值是与键对应的新值。本例中 [单词 , 1] 这样的流在经过 reduce_by_key 的处理后，最终会得到 [单词 , 出现次数] 这样的流。

最后通过管道把这个映射与 stdout 连接起来，stdout 就会打印出键与值的组合，这样就能在界面上看到单词与它的出现次数。这次没有做其他处理，需要的话，也可以在打印之前添加对单词的显示顺序进行排序的管道。

套接字编程

UNIX 中以流为基础设计的套接字（socket）当然也可以在 Streem 中使用。图 2-15 是使用套接字开发的最简单的网络服务——Echo 服务（原封不动地返回收到的消息）的程序。

```
# 打开8007端口提供服务
tcp_server(8007) | {s->
  s | s
}
```

图 2-15　Echo 服务

真简单。Streem 可以轻松编写出与流模型相匹配的程序。

我们先来看一下这个程序。tcp_server 函数打开指定端口的服务器套接字，等待客户端连接。Streem 的服务器套接字是客户端套接字的流。

客户端套接字是来自客户端的输入输出的流，所以下面这行代码的作用就是原封不动地返回收到的消息。

```
s | s
```

如果你想对收到的消息进行一些加工，可以在管道中间插入加工数据的流。

配管工作

正如我们前面看到的一样，管道的结构如下所示。

```
表达式 1 | 表达式 2... | 表达式 n
```

“表达式 1”是产生值的流（这里称为生产者），“表达式 2”的后面是对值进行转换及加工的流

（过滤器），最后的“表达式 n”是输出流（消费者）。

生产者中既有像 stdin 那样将外部输入作为流读取的类型，也有像 seq() 那样通过计算来产生值的类型。一旦在生产者的位置上放置函数表达式，生产者就会调用这个函数，生成 return 或 emit 返回的值。

过滤器在大多数情况下是函数，它会把接收到的值作为参数进行调用，再把 return 或 emit 返回的值传递给后面的流。

最后的消费者是只负责接收值而不对值进行 emit 的流。

Streem 的基础程序都会准备这种连接各个流的管道，通过流转来自生产者的值，从而对值进行加工，我们可以将这一系列处理形象地称为配管工作。虽然这种计算模型不是无所不能的，但优点是能够编写出抽象度高且易于理解的并发程序。有时放弃一些功能反而能够使产品更容易理解，该模型就是一个典型例子。

不过并非所有的程序都能够只用一条数据流来解决问题。对于那些不能用一条数据流解决问题的程序，如果将其全部放弃，未免过于激进，在这种情况下就需要更复杂一些的管道。具体来说，可以增加两个功能：把多个流整合为一个（合并）的功能，以及把一个流分割成多个来同时发送数据（广播）的功能。

如果可以设置流与流之间的缓冲区大小，那当然是再好不过的了。

管道的合并

在之前的例子中数据流都只有一条，非常简单，这一点很好，但并不能解决所有的问题。

有时还需要把多条管道合并为一条，或者把一条管道分拆为多条。管道合并时使用“&”符号。

```
管道 1 & 管道 2
```

这样就可以将来自“管道 1”的值与来自“管道 2”的值整合到一个数组中，产生新的管道。合并的管道中任意一个管道结束处理时，整个新管道就会随之结束。例如在之前的 cat 的例子的基础上增加行号显示（相当于 cat -n），代码如图 2-16 所示。

```
seq() & stdin | stdout
```

图 2-16 cat -n 的实现示例

运算符“&”的优先级高于“|”，所以

```
a & b | c
```

会被解释为

```
(a & b) | c
```

如果省略了 seq() 函数的参数，就会从 1 开始无限循环。stdin 从标准输入逐行读取数据并写入到管道，所以合并后会形成以下数组。

```
[行号, 行]
```

把这个数组写入 stdout，cat 就附带行号了。要想让程序更加实用，还需要进行行号对齐等格式方面的工作，只要在 stdout 之前放置格式化的管道即可[①]。

通道缓冲

末尾不是消费者的管道会返回“通道”（channel）这个对象。下面这行代码中的 sequence 就是通道，表示合并了 seq() 产生的数组与 stdin 的输入这两者的流。

```
seq() & stdin -> sequence
```

我们可以认为管道是用通道将进行流处理的“任务”连接起来而形成的。

当然，各个流的数据处理速度不尽相同。如果前段处理产生数据的速度过快，数据就会积压在通道里，耗费大量内存。相反，如果通道中完全不缓冲数据，前段处理的等待时间就会变长，导致效率低下。

于是我们让通道只缓冲适量的数据。不过真正合适的缓冲区大小是由程序决定的，我们无法完全推测出来。有时为了提升性能，需要明确指定缓冲区的大小，这时就需要用到 chan() 函数。

chan() 函数会显式地生成通道对象。如果管道运算符“|”的右侧是通道对象，就让该通道成为流输出的目的地。另外，把整数作为参数传给 chan() 函数时，该整数就是缓冲区的大小。图 2-17 在图 2-16 代码的基础上显式地指定了缓冲区的大小为 3。

```
seq() & stdin | chan(3) | stdout
```

图 2-17　指定了缓冲区大小的 cat-n

如果缓冲区的大小为 0，管道就会以前后交替的形式运行，产生一条数据后就要等到它被消费为止。在单核环境中这种方式或许会很方便。

① Streem 还在设计之中，格式化的相关细节尚未确定。

广播

比如在开发一个聊天服务时，需要把一个人发出的消息发送给全部参与者，在这种情况下也可以使用通道。如果将 chan() 函数生成的通道与多个流连接，那么向通道输入的值就会被广播到连接的所有流中。

图 2-18 是把图 2-15 的 Echo 服务改造为向全部参与者发送消息的 Chat 服务时的代码。

```
broadcast = chan()
# 打开8008端口提供服务
tcp_server(8008) | {s->
  broadcast | s    # 回复所有参与者的消息
  s | broadcast    # 将消息发送给所有参与者
}
```

图 2-18 Chat 服务

也许有人已经注意到了，广播通道是有状态的。这就意味着连接 broadcast 的流会作为目的地记录在 broadcast 中。另外，当发送目的地的流关闭，或者显式地用 disconnect 方法断开连接时，就需要把这个流从发送目的地的列表中去除。虽然 Streem 中要求对象不可变，但是为了使程序更易于理解，有时也需要做出一些妥协。当然，broadcast 内部对状态变化做了并发控制，所以在并行运行时也不会出现问题。

小结

本节我们设计了 Streem 这个以管道为计算模型核心的语言。如果程序非常适合使用流处理，那么编写起来会简单到让人吃惊。

其实 Streem 语言的设计才刚刚起步，在达到实用程度之前还有很多事情需要考虑，比如如何进行异常处理、如何准备用户自定义的流、如何定义“对象”等。随着软件规模的扩大，在语言上需要考虑的东西也会不断增多。

语言设计者常用的一个“借口”就是没有人会用这个语言去编写大规模的程序。除非这个语言不能用，否则这个借口是站不住脚的。

2-3 节将继续深入探讨 Streem 的设计，同时也思考一下如何去实现这门语言。

时光机专栏

语言名和语法都和预想的不同

本节是 2015 年 1 月刊中刊登的内容，我终于进入到了本书的重点部分——Streem 语言的开发。连载时使用的名字是 Stream，但这个名字容易招致混乱，因此在成书时修改了语言名。

语法跟之前也略有不同。具体的变化有右侧赋值符号由“->”变为了“=>”，以及函数表达式的参数部分由“{|x,y|x+y}”变为了“{x,y->x+y}”。另外，表示标准输入输出的 stdin、stdout 的名字也由大写变为了小写。我觉得不做任何修改直接给大家看连载时的内容倒也不失为一件乐事，但纠结了很长一段时间后，还是觉得当时的内容除了当历史资料以外没有太大用处，于是最终还是决定进行修改。

我在写这篇稿子的时候，语言处理器还完全没做好，所以本节的内容在当时只不过是还未实现的构想罢了。但为了避免语法前后矛盾、不能实现等，我写了最基本的 yacc 代码。当初只是将它作为备份上传到了 GitHub 上，没想到却成了 Hacker News 等海外新闻网站的热门话题，这对我来说也是一个美好的回忆。

最后有一点需要注意，在本节介绍的构想中，有一些功能在本书出版时还没有实现，比如在 Chat 服务中使用的 chan() 等。我会在以后去逐个实现这些功能，但是现在我无法保证什么时候能完成。

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

2-3 首先开发语法检查器

本节将开始实现2-2节中设计的以流为基础的语言。在验证完流模型的可行性之后，我们会先开发一个语法检查器来验证语法的可行性。终于要开始实现了，真是让人兴奋！

2-2 节介绍了使用基于流的计算模型的并行编程的语法。对于一般的并发处理来说，这些就已经足够了。下面我们先来研究一下这些语法的可行性。

任务构成模式

在基于消息的并发处理中，任务（或者线程和进程）的构成存在某种模式，以下是几种典型的模式。

- 生产者 – 消费者模式
- 轮询调度模式
- 广播模式
- 汇总模式
- 请求 – 应答模式

接下来我们就来看一下这些模式到底是什么样的，以及是如何应用在流模型中的。

生产者 – 消费者模式

生产者 – 消费者模式是通过消息进行并发处理的基本模式，由生产者生成数据，再传递给消费者（图 2-19）。最典型的例子就是 shell 的管道，不过这种模式在其他场景中也可以使用。



图 2-19　生产者 – 消费者模式

在流模型中，以生产者和消费者之间用流连接的形式编写代码。假定 P 为生产者，C 为消费者，用 Streem 语言编写出来的代码就是下面这种形式。

```
P | C
```

作为从生产者 – 消费者模式自然扩展的一种形式，有时 C 会对收到的数据进行加工而成为生产者，并向下一个消费者发送数据。在这种情况下，由于 C 对数据进行了加工，所以在 Streem 中就称为过滤器。管道的代码如下所示，当然也可以连接多个过滤器。

```
P | F | C
```

轮询调度模式

为了最大限度地利用多核，对生产者 – 消费者模式进行变形，准备多个消费者，让它们分担数据处理工作，这种轮流处理的模式就称为轮询调度（round robin）（图 2-20）。

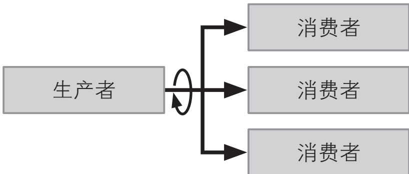


图 2-20　轮询调度模式

轮询调度的目的是分散压力，所以一般会准备多个进行相同处理的任务，让它们轮流进行数据处理。需要注意的是，如果在什么都不考虑的情况下让这些任务轮流进行数据处理，就会出现前面的数据处理一直不结束，后面的数据处理被迫等待的情况。虽说是轮流处理，但并不是将数据处理轮流分配给所有的任务，而是轮流分配给空闲的任务。在这方面我们还需要多费一些心思。

（多线程）Web 服务器就是一个轮询调度的例子。将客户端发来的请求分配给工作线程正是轮询调度的做法。

Streem 的语言处理器内置了支持轮询调度的结构。也就是说，如果按照普通的生产者 – 消费者模式去编写代码，Streem 就会自动创建多个任务进行处理。具体来说，只要准备好如下所示的流，Streem 就会根据 CPU 的内核数来创建同等数量的消费者进行处理。在内核数量增加时，即使不修改任何代码，也能在性能上得到提升。

```
P | C
```

广播模式

将同一条消息分配给多个任务进行处理的情形称为广播模式（图 2-21）。

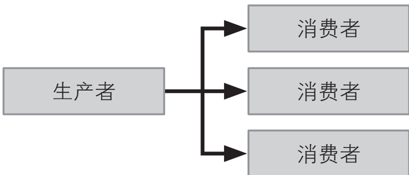


图 2-21　广播模式

广播模式的例子有聊天服务，其中一位成员的发言会被发送给聊天室里的所有成员。

Streem 中编写广播模式的方法是把多个流连接到一个流。假定 P 为生产者，C1 到 Cn 为消费者，将消息发送给所有成员的代码如下所示。

```
P | C1
P | C2
:
P | Cn
```

当所有消费者都保存在数组 ary 中时，代码如下所示。

```
ary.map{c -> P | c}
```

汇总模式

与一个生产者发送消息给多个消费者的广播模式相反，汇总模式是多个生产者把消息发送给一个消费者（图 2-22）。汇总模式的一个典型例子是日志收集，将各个地方产生的日志汇总到一个地方保存。

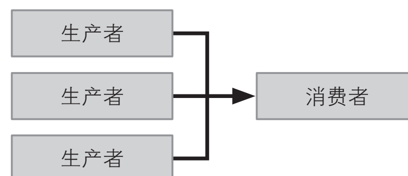


图 2-22　汇总模式

Streem 中汇总模式的实现与广播模式正好相反。假定 P1 到 Pn 为生产者，C 为消费者，代码如下所示。

```
P1 | C
P2 | C
:
Pn | C
```

当所有的生产者都保存在数组 ary 中时，代码如下所示。

```
ary.map{|p| p | C}
```

请求 – 应答模式

请求 – 应答模式简单来说就是发送消息给一个任务，再让这个任务把处理完毕的数据作为消息返回（图 2-23）。

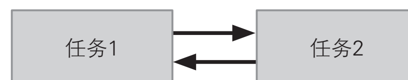


图 2-23　请求 – 应答模式

老实说，我并不推荐在并发编程中经常使用请求 – 应答模式。与其发送消息等待应答，还不如在同一个任务中进行处理更为高效，因为后者没有发送消息的开销。而要避免等待，就需要让发送请求和接收应答异步，这样就会使代码变得更加复杂，更加不易于理解。

因此，Streem 不支持请求 – 应答模式。这里推荐大家使用同步调用普通函数的方法来实现。

■ 开发编程语言的第一步

看起来 Streem 语言背后的流模型具备充分的可行性，那么我们就开始实现这门语言吧。是不是很兴奋呀？

大家在开发自己的语言时，首先会从哪里开始呢？

当然没人规定必须从哪里开始开发。我记得在二十多年前开发 Ruby 时，是从检查语法是否正确的语法检查器开始的，而在 2010 年开发 mruby 时，则是从虚拟机开始的。

从反复试验的地方开始开发

这么做是有原因的。在开始开发 Ruby 时，我还没想好用什么样的语法。作为编程语言迷，我在语言设计上最看重的就是语法。为了把语法确定下来，我进行了反复试验，并将这一工作作为优先事项。实际上，早期的 Ruby 语法与现在有着很大的差异，Ruby 语法是经过反复试验才成长为现在这个样子的。这是从语法分析器开始开发官方 CRuby 的最大原因。

而 mruby 一开始就定下来语法要与现有的 Ruby 保持兼容，所以在语法上没有什么反复试验的余地，于是我就把着眼点放在了内存使用量少、运行效率高的虚拟机的开发上。因此，mruby 的开发是从对虚拟机的结构和指令集进行各种尝试开始的。最开始的时候是手敲字节码（虚拟机的命令序列），然后直接用在虚拟机上。在虚拟机可以在某种程度上运行起来之后，我又基于从 CRuby 复制过来的语法定义开发了语法分析部分和代码生成部分。

总之，秘诀就是先开发最需要进行反复试验的部分。为了进行反复试验，就需要在早期采取一些行动。在开发 CRuby 时，我首先开发了检查 Ruby 程序的语法是否正确的语法检查器来进行反复试验。而在开发 mruby 时，则首先开发了运行手敲的示例代码的字节码（一个计算阶乘的程序）的虚拟机。

Streem 语言也从语法检查器开始开发

那么 Streem 语言要从哪里开始开发呢？

实现 Streem 时有很多需要反复试验的地方。Streem 这门新的语言有新的语法，所以我还是很在意语法质量的。另外，Streem 的精髓在于支持多核环境下抽象度高的并发编程，因此负责这部分的任务调度器的开发难度也比较高，预计需要反复试验。

再三考虑之后，我决定和 CRuby 一样从语法检查器的实现开始。

虽说如此，但与开发 CRuby 的时候不同，现在的我已经熟悉 yacc 的用法了，所以开发起来应该比以前更加顺畅。

首先是创建项目。这里我采用现在比较流行的一种做法——使用 GitHub 进行代码管理。首先登录到 GitHub，创建一个空的软件仓库，名为“matz/streem”。

将这个新创建的软件仓库复制到本地环境（图 2-24）。这里需要先准备一个 README 文件来展示这是一个什么项目。GitHub 中推荐使用 Markdown 格式，所以我们就新建一个使用 .md 扩展名的 README.md 文件（图 2-25）。

开发就从这里开始。

```
$ git clone git@github.com:matz/streem.git ↵
```

↵在这个标记处换行

图 2-24　从 Github 获取软件仓库

```
# Streem - stream based concurrent scripting language

Streem is a concurrent scripting language based on programming model
similar to shell, with influence from Ruby, Erlang and other
functional programming languages.
In Streem, simple `cat` program is like this:

   stdin | stdout

# Note

Streem is still under design stage. It's not working yet.  Stay tuned.

# License

Streem is distributed under MIT license.

Copyright (c) 2015 Yukihiro Matsumoto
```

图 2-25　README.md

软件构成

目前 Streem 语言处理器（只有语法检查器）的构成如图 2-26 所示。

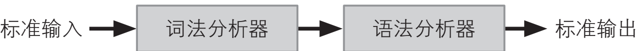


图 2-26　Streem 语言处理器的软件构成

首先将词法分析器读入的程序转换为被称为“单词”的记号序列。单词是指由关键字、数值、字符串和运算符等多个字符串组成的有意义的块。

语法分析器对单词序列进行解析，检查表述是否符合语法（不符合的话就报错），语法正确的话就执行符合语法含义的动作。目前 Streem 还只有语法检查器，将来肯定还要构建程序结构以实现运行，预计还要生成语法树和字节码，并把这些信息发送给运行部分（虚拟机等）去实际运行。

接下来我们看一下词法分析器和语法分析器的开发。

词法分析器的开发

词法分析器有多种实现方法，这次我们使用 yacc 的搭档，即词法分析器的自动生成工具 lex（正确来说是 GNU 扩展版的 Flex）。Ruby 的词法分析器是自己开发的，但是我经过各种研究发现，lex 也可以在很大程度上实现词法分析器。如果能用工具的话还是使用工具更让人放心。如果有一天 Streem 在功能或者性能上出现了问题，也许就会像 Ruby 那样换成自己开发的词法分析器，不过现在还不需要担心这一点。

准备好词法定义文件后，lex 会自动帮我们生成进行词法分析的 C 函数，具体来说就是在用 `yylex()` 调用时会返回“下一个单词”的函数。

后面介绍的 yacc 生成的语法分析器在获取下一个单词时会调用 `yylex()` 函数，这实在是太方便了。如果自己编写词法分析器，就需要编写 `yylex()` 函数了。

语法定义的编写

我们先来实现语法检查器。语法分析器有多种实现方法，像 Streem 这种规模的语法，用“递归下降语法分析法”自己写一个语法分析器就足够了。不过没必要非得自己开发，所以这次我们就用 yacc 来实现语法分析器。首先准备 parse.y 文件。

这个文件是刚才提到的工具 yacc 处理用的源代码。如果要详细介绍 yacc 的语法，得写一本书才能讲完，所以这里就不详细介绍了。为了方便大家理解，我来介绍一下最基本的内容。1-2 节中也介绍过这部分内容，这里正好复习一下。

lex 定义的语法

作为 lex 的输入的定义文件通常使用扩展名“`.l`”。图 2-27 是 lex 定义文件的示例。

定义文件大体上可以分为以下 3 个部分，各部分用“`%%`”隔开。

- 声明部分
- 词法定义部分
- C 语言部分

声明部分声明 lex 中使用的选项等。在声明部分编写的以“%{”开始以“%}”结束的代码会被直接插入到生成的 C 语言代码中。我们可以在这里编写头文件引用、变量和函数原型等声明。

词法定义部分用于编写表示单词的正则表达式以及支持的动作。

最后的 C 语言部分用于定义词法解析器中使用的 C 语言函数等。这一部分也会被直接插入到生成的 C 语言代码中。在语法定义部分的动作中使用的函数等多在这里定义。

把图 2-27 的文件（假定文件名为 lex.l）传给 lex 命令进行处理之后，会生成一个名为“lex.yy.c”的文件。实际上这里使用的不是原版的 lex，而是 GNU 扩展版的 lex——flex。运行步骤如图 2-28 所示。

```
/* 声明部分为空 */
%%
"+"             return ADD;
"-"             return SUB;
"*"             return MUL;
"/"             return DIV;
"\n"            return NL;

(([1-9][0-9]*)|0)(\.[0-9]*)? {
    double temp;
    sscanf(yytext, "%lf", &temp);
    yylval.double_value = temp;
    return NUM;
};

[ \t] ;

. {
    fprintf(stderr, "lexical error.\n");
    exit(1);
}
%%
/* C语言部分也为空 */
```

图 2-27　lex 定义文件的示例

yacc 脚本的语法

词法分析之后是语法分析。yacc 是用于生成语法分析器的工具，全称是 Yet Another Compiler Compiler。

yacc 是在 20 世纪 70 年代开发的用于生成编译器语法分析部分的工具，据说当时开发了很多这种被称为“编译器的编译器”的工具。其中，后被开发出来的 yacc 取“现有工具的补充”之意，命名为“yet another”。但讽刺的是，在当时开发的那么多编译器的编译器中，现在只有 yacc 还在被使用。

说来也怪，似乎只要用“yet another”命名就会生存下来。Ruby 的虚拟机中也是只有 YARV（Yet Another Ruby VM）生存了下来。

```
$ flex calc.l ↵
$ head lex.yy.c ↵ ←只显示前面10行

#line 3 "lex.yy.c"

#define  YY_INT_ALIGNED short int

/* A lexical scanner generated by flex */

#define FLEX_SCANNER
#define YY_FLEX_MAJOR_VERSION 2
#define YY_FLEX_MINOR_VERSION 5
```

图 2-28　flex 运行步骤

作为 yacc 的输入的定义文件一般使用扩展名“.y”。与 lex 相同，定义文件大体上可分为以下 3 个部分，各部分用“%%”隔开。

- 声明部分
- 语法定义部分
- C 语言部分

声明部分声明 yacc 中使用的单词、运算符的优先级和规则的数据类型等。另外，C 语言子程序的声明也可以包含在内。与 lex 相同，声明部分中以“%{”开始以“%}”结束的部分会被直接插入到生成的 C 语言代码中。我们可以在这里编写头文件引用、变量和函数原型等声明。

C 语言部分定义语法分析器中使用的 C 语言函数等。这一部分也会被直接插入到生成的 C 语言代码中。语法定义部分的动作中使用的函数等多在这里定义。

用巴科斯范式定义语法

在语法定义部分，我们可以用类似于巴科斯范式（Backus Naur Form，BNF）的规则来编写语法分析器能理解的语法。巴科斯范式是巴科斯（Backus）和诺尔（Naur）在定义 Algol 语言的语法时发明的写法规则（图 2-29）。

```
expr        : NUM
            | expr '+' NUM
            | expr '-' NUM
            | expr '*' NUM
            | expr '/' NUM
            ;

val         : NUM
            | '(' expr ')'
            ;
```

图 2-29　巴科斯范式示例

我们来看一下图 2-29 的例子所表达的含义。数学式（expr）是以下式子中的一种。

- 值
- 数学式 + 值
- 数学式 – 值
- 数学式 * 值
- 数学式 / 值

值（val）是以下式子中的一种。

- 数值
- '(' 数学式 ')'

该例子虽未涉及，但巴科斯范式实际上可以编写个别规则所支持的动作。动作是用“{ }”括起来的 C 语言代码，用于编写匹配到规则时要进行的处理，比如生成语法树、生成代码等。

根据图 2-29 定义的规则，我们来分析一下下面这个程序。

```
1 + 2 + (3 * 4)
```

结果如图 2-30 所示。

运行 yacc 后会生成一个 y.tab.c 文件。但原版的 yacc 有诸多限制，比如无法生成多个线程安全的语法分析器等，所以就使用扩展版本的 GNU bison。

我们只需运行如下命令即可。

```
$ bison parse.y
```

实际上，在 Ubuntu 等很多 Linux 发行版中，即使启动 yacc，实际运行的也还是 bison。

```
● 正确的语法

1 + 2 + (3 * 4)

  1 -> 数值 -> 值 -> 数学式
  2 -> 数值 -> 值
  1 + 2 -> 数学式 + 值 -> 数学式
  3 -> 数值 -> 值 -> 数学式
  4 -> 数值 -> 值
  (3 * 4) -> (数学式 * 值) -> (数学式) -> 值
  1 + 2 + (3 * 4) -> 数学式 + 值 -> 数学式

● 错误的语法

1 - * 2

  1 -> 数值 -> 值 -> 数学式
  * -> 无匹配规则（错误!）
```

图 2-30　巴科斯范式解析

Streem 的语法

终于到了 Streem 的语法定义环节了，但遗憾的是因版面所限，不能在本书中展示 Streem 语法定义的全部内容。

想看相关代码的读者请移步 GitHub，Streem 的源代码在 github.com/matz/streem 上。我在本次解说相应的地方打了下面这个标签，大家可以参考。

```
201502
```

虽然不能对全部代码进行说明，但我还是来说一下 Streem 的基本设计方针。

Streem 是参考 mruby 的源代码开发的，所以处处都能看到 mruby 的影子。

但与语法复杂的 Ruby 不同，Streem 的语法被控制在较小的规模，比如不能在字符串中嵌入表达式，也没有“Here 文档”的功能。

在设计 Streem 时，相较于钻研语法，我认为在保持简洁的同时探索流模型的可能性更为重要。至少在现阶段是这样的。如果将来 Streem 能发展到实用的程度，那么在这个过程中它的语法或许会得到强化。

下一节我们将继续开发 Streem 的语言处理器。

小结

为了展示流模型在并发编程中的可行性，本节介绍了如何用流模型编写各种任务构成模式。

此外，作为开发编程语言的第一步，我们开始了语法检查器的开发。使用 yacc 和 lex 工具可以比较简单地开发编程语言。

下一节我们将思考语言运行的相关内容。另外，在用语法检查器进行测试的过程中，我开始不满于最开始设计的 Streem 语法，关于这部分内容，我们也会在下一节进行探讨。

时光机专栏

被颠覆的开源的常识

本节是 2015 年 2 月刊中刊登的内容。虽说只讲了语法检查器，但我终于开始了语言处理器的开发，这让我感到热血沸腾。

在其他章节中也提到过，我将这里开发的语法检查器的源代码（200 行左右的 parse.y）上传到 GitHub 之后，得到了全世界的关注，星数也超过了 1000。不过我之后的开发进度较慢，也许大部分人都等得不耐烦了。

我从事自由软件开发工作已经超过了 25 年，但这次创建 Streem 时发生的事情还是远远超过了我的预期。创建开源软件，如果公开时源代码不能正常运行，就得不到人们的关注，可如果代码过于复杂，就会让其他人难以下手，对社区的发展不利。这是一种很微妙的平衡。

可是在 Streem 被公开时，别说正常运行了，当时只有一个让人难以想象出是什么语言的语法检查器，但即便如此，它还是得到了很大的关注。当然这可能都“归功于”我是 Ruby 的开发者，但光凭知名度就能颠覆开源的常识，这一点还是让我很惊讶。

关于 GitHub 的标签我再补充几句。标签是按照杂志连载时的月份来打的。比如，本节是 2015 年 2 月刊中刊登的内容，所以打的标签就是 `201502`。整理成书时没有对标签进行修改，所以代码还是当时的样子。如果有读者通过标签去查看当时的代码，那么请注意本书与当时的源代码有些许不一致的地方。

2-4 事件循环

本节将继续Streem语言的实现。海外的新闻网站也对Streem进行了介绍，老实说，这让我这个作者感到很意外。本节将首先对Streem的语法进行一些改良，然后开始Streem的核心——流运行的原型开发。

从完成原稿到杂志上架销售，中间要经过编辑、校对和印刷等很多工序。《日经 Linux》在每月 8 号左右上架，但原稿的撰写在很早之前就开始了。大家在《日经 Linux》上读到的文章，都是我在杂志上架一个月前写的。

我在写 2-3 节的内容时编写了 Streem 的语法检查器，简单地写了一下 README 之后就和语法检查器一起上传到了 GitHub 上。结果被眼尖的人看到，Streem 的相关消息就在 SNS 等社交网络平台上不断扩散开来。这是 2014 年底的事情了，那时距离当期杂志上架还有一个月。

尽管还不能运行，但依旧有人在 GitHub 上发送 Pull Request 给我，这让我很感动。

没想到一个只能做语法检查的原型，竟能在 Hacker News 等网站成为话题。因为本来是要在 1 月 8 号上架的 2015 年 2 月刊的解说用的代码，所以 Copyright 写成了 2015 年，这一点也被眼尖的人指出来了，真是让人出乎意料，看来“写 Ruby 的那个松本又开发新语言了”这件事情比我想象的更具话题性。

我长期参与开源活动，一直以来都认为开源软件的流行离不开社区和可以运行的代码，但这次开发 Streem 的经历让我有了新的认识。即使只是原型，即使还不能运行，只要有话题性，只要能打开这个话题，开源软件就可以活跃起来。

■ 动手实现核心部分

语法已经基本确定下来了，这次我们来研究一下其他方面。语法分析器之后应该是代码生成和虚拟机的实现，但这次我选择先去开发 Streem 实现的核心——流运行的原型。

我设想的 Streem 程序如下所示。

```
stdin | {x->x.toupper()} | stdout
```

运行该程序，就会构建出一个由以下 3 个任务组成的管道。

- 从标准输入读取 1 行

- 在读取的行中处理函数
- 将处理结果写入标准输出

最后运行管道处理。

首先来开发构建和运行管道的 C 程序的原型。

main 的原型

C 程序 main 函数的原型如图 2-31 所示。

```
int
main(int argc, char **argv)
{
  strm_stream *strm_stdin = strm_readio(0 /* stdin*/);
  strm_stream *strm_map = strm_funcmap(str_toupper);
  strm_stream *strm_stdout = strm_writeio(1 /* stdout */);

  /* stdin | {x->x.toupper()} | stdout */
  strm_connect(strm_stdin, strm_map);
  strm_connect(strm_map, strm_stdout);
  strm_loop();

  return 0;
}
```

图 2-31　main 函数的原型

首先，表示管道组成部分的结构体是 strm_stream。main 函数的开头对构成这次原型管道的 3 个组成部分进行了初始化。

接着，strm_connect() 函数把这 3 个 strm_stream 连接了起来。运行 Streem 程序时，程序内部会进行这种管道的构建处理。

最后，strm_loop() 函数运行管道处理。处理完毕后，strm_loop() 运行结束，程序运行完成。

管道的构造

表示管道的结构体 strm_stream 的定义如图 2-32 所示。

```
struct strm_stream {
  strm_task_mode mode;  /* 生产者、过滤器、消费者中的一个 */
  unsigned int flags;   /* 标志 */
  strm_func start_func; /* 开始函数 */
  strm_func close_func; /* 后续处理函数 */
  void *data;           /* 流本身的数据 */
  strm_stream *dst;     /* 输出目的地的流 */
  strm_stream *nextd;   /* 输出目的地的链接 */
};
```

图 2-32　strm_stream

strm_stream 的结构体被 dst 指针链接起来组成了管道（图 2-33）。当一个流被多个流连接时，可以通过 dst 的输出目的地的链接字段 nextd 来访问这些流（图 2-34）。像这样一个流分为多个，或者多个流访问一个流，就形成了流的网状结构（尽管大部分情况下是串联的），这种网状结构就称为管道。

Streem 程序的本质就是这种流的定义以及管道的构建。之后就会以事件在所构建的管道中流动的形式进行处理。

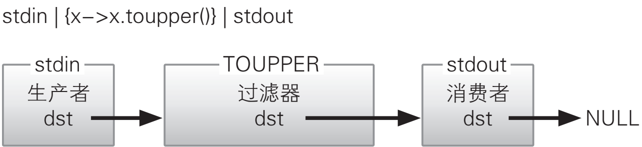


图 2-33　管道（cat 的例子）

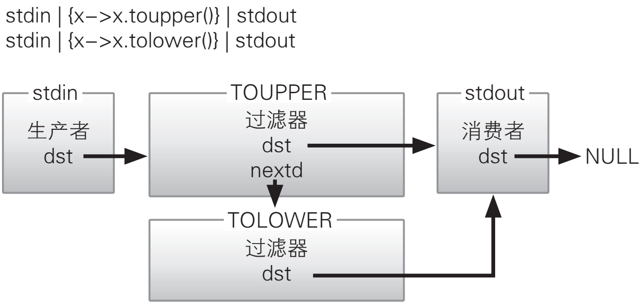


图 2-34　多个流的结合

自行开发事件循环

实际进行事件处理的是 strm_loop() 函数，这个函数需要进行以下处理。

- 根据 I/O 的输入等产生事件
- 对产生的事件进行相应的处理。如果是输入事件，则进行数据的读取、行的分割等处理
- 把事件处理的结果发送到管道的下一个流，进行接下来的处理
- 之后进行循环

世界上有很多进行这种事件处理的库，光是知名的就有以下几种。

- libevent

- libev
- libuv

libevent 是老牌的事件处理库，是使用回调实现伴有 I/O 等的事件处理的一系列库的鼻祖。

libev 对 libevent 做出了一些改善。libev 与 libevent 的 API 不同，主要改善了速度以及废除了对一个文件描述符进行检测的 watcher 数量的限制。

libuv 是在 libev 的基础上为 node.js 开发的事件处理库。它最与众不同的地方就是在 UNIX 之外的操作系统（也就是 Windows）上工作。另外，线程相关的 API 接口也很丰富。

一开始我考虑让 Streem 也使用这些库来实现，可在多线程的地方出现了问题。我想在 Streem 的实现中使用多线程，可是这些事件处理库不支持多线程，准确来说是不支持在线程间传递事件。另外，Streem 中会大量使用 seq() 这种非 I/O 的事件，在这一点上也遇到了问题，所以我决定自己去实现事件循环。

不过，也可能是因为我有些东西不知道所以才碰到了这些问题。我真的太久没有接触事件驱动编程了，虽然开发过早期的 Ruby 线程系统，但其实也没什么多线程编程的经验。

话说回来，我在大学毕业后进入的那家公司独自开发了 X Window 工具箱，那时候用到了事件驱动编程。

所以今后我还是会继续进行事件处理库的相关研究。因为如果有什么方法可以满足需求的话，肯定还是“不要重复发明轮子”为好。

I/O 事件的检测

在事件驱动编程的输入输出中，重要的是避免“阻塞”。这里的阻塞是指，如果在数据还没到达文件描述符的时候读取数据，那么系统调用就会在数据到达之前一直处于停止状态，所以在实际读取数据之前，我们需要知道数据是否已经到达文件描述符。

有几种方法可以做到这一点，其中最老套的就是 select 系统调用（图 2-35）。但 select 有一些缺陷，那就是等待的文件描述符有数量限制，以及检查 n 个文件描述符需要花费 n 倍的时间。这些限制导致 select 不适合用来处理近年来备受重视的大量访问的场景。于是，更好的系统调用便应运而生。

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

/* select系统调用
   nfds - 等待的最大文件描述符
   readfds - 等待读取的文件描述符集合
```

```
   writefds - 等待写入的文件描述符集合
   exceptfds - 等待异常的文件描述符集合
   timeout - 最大等待时间（或者设为NULL，表示不启用超时一直等待I/O到来） */
int select(int nfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);

/* fd_set的初始化 */
void FD_ZERO(fd_set *set);
/* 将fd加入到fd_set集合中 */
void FD_SET(int fd, fd_set *set);
/* 测试fd是否在fd_set集合中 */
int  FD_ISSET(int fd, fd_set *set);
/* 将一个fd从fd_set集合中清除 */
void FD_CLR(int fd, fd_set *set);
```

图 2-35　select 系统调用

“更好的系统调用”包括 Linux 的 `epoll` 系统调用和 BSD 系操作系统的 `kqueue` 系统调用。遗憾的是这些系统调用还没有被标准化，需要根据操作系统进行切换。实际上前面提到的事件处理库 `libevent`、`libev` 和 `libuv` 在内部就区分使用了这些系统调用。

epoll 系统调用

我决定将 `epoll` 系统调用用在这次的原型中。`epoll` 虽然只能在 Linux 系统中使用，但本文是在《日经 Linux》杂志上连载的，所以这一点（至少在我撰稿的时候）不用担心。不过 Streem 语言的最终实现还需要依靠 `epoll` 之外的 I/O 检查。

`epoll` 系统调用（实际上是 `epoll_create`、`epoll_ctl` 和 `epoll_wait` 这 3 个系统调用）的使用方法如图 2-36 所示。

```
#include <sys/epoll.h>

/* 为epoll创建文件描述符 */
int epoll_fd = epoll_create(10);

struct epoll_event ev;
ev.events = EPOLLIN;
ev.data.ptr = data; /* 把这个数据传给epoll_wait */
/* 添加要等待的文件描述符fd到epoll */
/* EPOLLIN是等待读取 */
```

```
epoll_ctl(epoll_fd, EPOLL_CTL_ADD, fd, &ev);

struct epoll_event events[10];
for (;;) {
  /* nfds - 数据已到达的fd的数量 */
  int nfds = epoll_wait(epoll_fd, events, 10, -1);
  for (int i=0; i<nfds; i++) {
    /* 拿到epoll_ctl传过来的数据 */
    data = events[i].data.ptr;
    ...
  }
}
```

图 2-36　epoll 系统调用

与 `select` 系统调用不同，在 `epoll` 中，事件信息会被传递给结构体，所以不需要考虑等待的 I/O 的数量。另外，`epoll_ctl` 与 `epoll_wait` 即使在不同的线程中也能保证正常工作，所以可以在一个线程中用 `epoll_wait` 构建事件循环，在其他线程添加要等待的 I/O。

事件队列

Streem 原型的系统构成如图 2-37 所示。

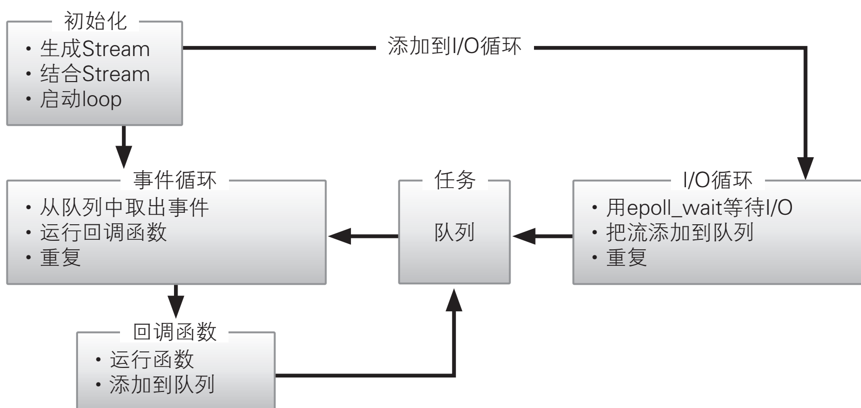


图 2-37　Streem 系统构成

首先在初始化阶段生成流（`strm_streem`）。之后，流被结合到一起构成管道。这里使用 `epoll` 把等待 I/O 的流添加到 I/O 循环。

接下来，I/O 循环会作为独立的线程启动。这个线程使用 `epoll_wait` 等待 I/O，把传来数据的流添加到任务队列中。添加到任务队列中的信息有 3 个，分别是运行对象的流、运行内容的函数

指针、前面的流传来的数据（void *）。管道开头的流中没有数据传来，所以传递的是 NULL。

之后在主线程中运行事件处理循环。在主线程的事件循环中会依次从任务队列取出信息，运行函数。

函数中会进行输入的读取、字符串化、传来的数据的加工或输出等实际处理。函数中使用 strm_emit() 向下一个流传递数据。

strm_emit() 函数有 3 个参数：第 1 个是与当前任务相对应的 strm_stream 结构体（注意不是数据传递目的地的流。这个参数的作用在于即使管道构造发生变化也不需要修改函数）；第 2 个是接收到的数据（void * 形式）；第 3 个是收到数据后，接着这个函数执行的回调函数。

即使调用了 strm_emit()，也并不意味着下一个处理就会被立刻执行。被传递的数据首先要添加到任务队列。函数运行结束后从主循环中取出下一个任务执行。

处理函数的模式

有了这种任务队列，不用太依赖循环就可以实现 Streem 的处理函数。处理函数的实现模式大体可分为以下 3 种。

1. 使用回调实现事实上的循环

这个模式用于实现输入处理这种生成数据的流。把处理分割为多个 C 函数，在每次进行 strm_emit() 时，设置回调函数为“下一个函数”。把回调设置为同一个函数就形成了事实上的循环。在原型的源代码中，io.c 的 read_cb 和 readline_cb 就使用了这个模式。

2. 直接处理接收到的数据

该模式只是逐个处理接收到的数据，而不会重复自身函数的处理。main.c 的 map_recv() 和 io.c 的 write_cb() 就属于这种模式。该模式会把 strm_emit() 的回调设置为 NULL。

3. 在循环中进行 emit

按理说在循环中进行 strm_emit() 也是可行的。这种做法虽然易于理解，但会把任务积压在队列中，在函数处理结束返回到主循环之前处理都不会有什么进展，所以不推荐使用这种模式。

原型的代码

这里讲解的原型的源代码放在了 http://github.com/matz/streem 代码库的 lib 目录中。在这个目录中执行 make 命令，就会生成一个名为 a.out 的可执行文件。这个可执行文件的作用在于把标准输入转为大写字母并输出。因为出版存在时间差，这里的内容可能会与将来的最新版有所出入，所以我给本节撰稿时的代码打了 201503 的标签。

今后的方向

虽然还在创建原型的试验阶段，但我已经渐渐看清了今后的方向。

首先要做的就是解决因使用 `epoll` 而只能在 Linux 上运行这一问题。可以想到的办法有大幅改写 `libuv`，或者在没有 `epoll` 的环境中使用 `kqueue` 和 `select` 等[①]。

其次就是实现多线程化。这次实现了一个等待 I/O 的线程和一个处理事件队列的线程。为了最大程度地利用多核，我希望根据 CPU 内核数生成线程来分担处理。

其实在准备本次原稿时，我尝试过用多个线程从事件队列中取出事件，但由于处理时间的关系，发生了运行顺序错乱的情况。如果放任不管，在输出文件内容时，输出顺序就会根据行的长度而发生变动。不管怎么说这都是一个大问题，所以这次就用一个线程来进行事件处理。把分配线程的单位由各个事件修改为各个管道或许能够解决这个问题，但这次已经没有时间进行尝试了。

小结

作为原稿素材诞生的 Streem 语言出乎意料地得到了广泛的关注，不过只是停留在原型这一层面上是没有意义的，因此我打算今后（与连载一起）继续完善，使其达到实用的程度。在杂志连载中实时讲解编程语言的开发过程是一次非常难得的体验。还请各位读者继续期待后面的内容。

时光机专栏

并发编程之难

本节是 2015 年 3 月刊中刊登的内容。这次我们开发了 Streem 语言处理器的核心——事件循环部分。

在 Streem 中，事件循环与词法分析、语法分析等进行语言处理的部分同等重要。事件循环并不容易实现，本来我打算使用现有的库，但没有一个库可以支持 Streem 要求的那种多线程，无奈之下我只好自己动手开发。这次开发的是单线程版本。

在下一节，我将把它升级为多线程版本，但是问题层出不穷。老实说，这本书涉及并发编程的部分都是反复试验和失败的记录。虽然听起来像是在为自己辩解，但对于我这种疏忽大意的人来说，并发编程太过深奥。正因为如此，我们才需要 Streem 那样的语言。它抽象程度高，可以帮助我们掩盖掉并发编程的难度。然而没有人来做这件事，所以我只好自己去做。

① 写完原稿后有人发来了使用 `select` 的 Pull Request，现在已经可以在 Windows 和 Mac 上运行了。

这次除了事件循环，我还对语法进行了修改。不过为了让本书更易于理解，我在正文里去掉了关于阶段性修改语法的内容。但另一方面，我觉得作为语言设计的资料，修改语法的原因非常有价值，所以最后就以专栏的形式附在了正文的后面。比如，我将下面这个语法修改成了 `{x->...}` 的形式。

- 函数中要像 `{|x|...}` 这样把参数用 `||` 围起来

另外，为了今后使语法变得更加简单，连载时“`if` 语句的条件表达式不带括号”的语法也变成了条件表达式带括号的形式。

改善语法的理由

在实现了语法检查器，又写了几个Streem语言的示例代码之后，我发现了一些不太满意的地方。

Streem从Ruby那里继承了代码块（匿名函数）的参数带“|”的写法，比如下面的代码。

```
{|x| x * 2}
```

但是Streem中的“|”符号多用来连接流，比如2-3节中就出现了下面这样的例子。

```
ary.map{|c| P | c}
```

这样就会造成混乱，老实说也并不美观，于是我稍微调整了一下匿名函数的语法，新的语法如下所示。

```
{ x -> x * 2 }
```

在没有参数或者省略参数时，所有“->”都可以省略不写。

```
{ print "hello\n" }
```

这些语法参考了Groovy和Swift。是不是很有现代感？

这样一来，语法就美观一些了。至少对我而言，语法是否美观会影响编程的激情，所以说这是一个很好的修改。

发生了语法冲突

但是这个修改造成了语法冲突。Streem 把 `if` 等语句的代码用“{ }”括起来，这就和匿名函数发生了冲突。准确来说，如果在函数调用后面加上代码块，匿名函数就会作为最后一个参数传给函数。这个语法规则导致了冲突的发生。

可能有人不理解语法冲突是怎么一回事。当语法分析器去读取被词法分析器分割为单词的程序，并用语法规则去解析时，无法决定该用哪个规则进行解析的情况就称为“冲突”。

拿下面这段代码来说，“`foo(x)`”之后出现的是“{”，这样就无法判断这是表示 `if` 语句的代码，还是表示传给 `foo` 函数的匿名函数了。

```
if foo(x) {
  print("hello\n")
}
```

3 种解决方法

解决这个冲突的其中一个方法是像 C 语言那样把 `if` 语句的条件表达式用括号括起来。

```
if (foo(x)) {
  print "hello\n"
}
```

这样一来，函数调用和 `if` 语句的代码就可以被明确地区分开来。不过在现代语言中，`if` 语句的条件表达式一般都不带括号（如 Swift 和 Go 等），而且我们也不希望代码量增加。

另一个办法是，既然匿名函数以“{”开始导致无法与 `if` 语句区分开来，那就使用其他符号来表示匿名函数。比如像 Ruby 的 lambda 表达式一样，匿名函数用“`->`”开头。这个方法怎么样呢？采用这个方法，匿名函数就会变成下面这样。

```
-> x { x * 2 }
```

函数调用代码如下所示。

```
map-> x { x * 2 }
```

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

这么表示匿名函数看起来也还不错，但函数调用的形式看起来不像是控制结构。我觉得这样就失去了Ruby代码块的美观性。其实过去在Ruby中用“->”代表匿名函数时，就考虑过引入这种代码块的语法，但我不是很喜欢，所以最后就没有采用。

最终我决定用第3种方法来解决这个问题，也就是不允许在不改变匿名函数和函数调用的语法的情况下，在if语句的条件里把匿名函数添加到函数调用中，具体做法如图2-A所示。

复制语法规则的代码从实现的角度来看并不是最好的方法，但这也是为了实现自己满意的语法而付出的代价，所以我也就想开了。

其实2-3节介绍的语法也会出现这个问题，但我却没有发现，这是为什么呢？原来是我一时疏忽忘记写把代码块传给函数的规则了。

```
/* 避免语法冲突的部分（摘录） */

/* 一般的表达式 */
expr      : expr op_plus expr
          | expr op_minus expr
          | expr op_mult expr
          | expr op_div expr
          | expr op_mod expr
          | expr op_bar expr
          | expr op_amper expr
          | expr op_gt expr
          | expr op_ge expr
          | expr op_lt expr
          | expr op_le expr
          | expr op_eq expr
          | expr op_neq expr
          | op_plus expr                 %prec '!'
          | op_minus expr                %prec '!'
          | '!' expr
          | '~' expr
          | expr op_and expr
          | expr op_or expr
          | primary
          ;

/* if等语句的条件表达式（差不多都是expr的副本） */
condition : condition op_plus condition
          | condition op_minus condition
          | condition op_mult condition
```

```
            | condition op_div condition
            | condition op_mod condition
            | condition op_bar condition
            | condition op_amper condition
            | condition op_gt condition
            | condition op_ge condition
            | condition op_lt condition
            | condition op_le condition
            | condition op_eq condition
            | condition op_neq condition
            | op_plus condition            %prec '!'
            | op_minus condition           %prec '!'
            | '!' condition
            | '~' condition
            | condition op_and condition
            | condition op_or condition
            | cond
            ;

/* 基本表达式（primary）的共同部分 */
primary0    : lit_number
            | lit_string
            | identifier
            | '(' expr ')'
            | '[' args ']'
            | '[' ']'
            | '[' map_args ']'
            | '[' ':' '}'
            | keyword_if condition '{' compstmt '}'
 opt_else
            | keyword_nil
            | keyword_true
            | keyword_false
            ;

/* 调用无代码块的函数 */
cond        : primary0
            | identifier '(' opt_args ')'
            | cond '.' identifier '(' opt_args ')'
            | cond '.' identifier
            ;
```

```
/* 调用有代码块的函数 */
primary    : primary0
           | block
           | identifier block
           | identifier '(' opt_args ')' opt_block
           | primary '.' identifier '(' opt_args ')
' opt_block
           | primary '.' identifier opt_block
           ;
```

图 2-A 避免冲突的语法

2-5 多线程与对象

本节继续实现Streem的核心部分。这次我们挑战通过线程来有效利用多核CPU。为了尽可能地让任务并行处理，我们只启动与CPU内核数同等数量的工作线程来运行任务。

2-4 节最后提到了如果用多线程运行任务，就会出现任务运行顺序错乱的情况。

多线程化

我们再稍微思考一下发生了什么。首先回顾一下之前的内容：lib 目录下的原型（a.out）从标准输入读取字符串，将其全部转换为大写，然后写入到标准输出。

```
% echo foo | a.out
FOO
```

2-4 节介绍的单线程版本的程序结构（准确来说是有一个等待输入输出的 I/O 线程和一个实际进行处理的工作线程）如图 2-38 所示。

当 I/O 线程检测到有可以输入的数据，或者工作线程向接下来的任务 `emit` 数据时，数据的处理内容就会被送往任务队列。工作线程从任务队列中依次取出任务，继续进行处理。

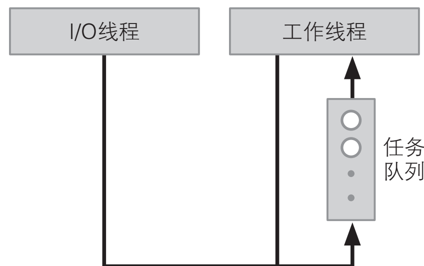


图 2-38　单线程版本

尝试多核化

我考虑把这个过程多线程化。为了最大限度地利用多核 CPU，我打算启动与 CPU 内核数同等数量的工作线程，让空闲的工作线程从任务队列中取出任务进行处理（图 2-39）。和之前一样，在任务运行中发生

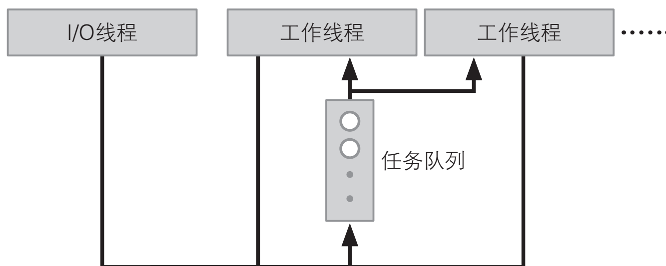


图 2-39　多线程版本（第 1 版）

emit 时，由于要向管道中的后续处理传递数据，所以要将任务放到队列中。另外，如果任务还有后续处理，就要把自身（的后续处理）加入到队列中。

启动这种结构的程序后，在进行短的输入或者从键盘进行标准输入时就会很顺畅。但是当输入一定程度的长文件时，就会出现异常，输出的转换结果与输入的文件的行的先后顺序发生了改变。这是不可以的。

我考虑了一会儿，终于明白了其中的原因。将输入的数据转换为 Streem 的字符串，或者将字符串中的字符转换为大写，在进行这样的处理时，行越长处理时间就越长。因为速度越快的工作线程会越先从任务队列中取出任务开始处理，于是在处理较长的行的线程还在工作时，后面处理较短的行的线程可能就赶在它前面结束了。多线程环境不能保证顺序和时间，像我一样只开发单线程程序的程序员经常忘记这一点。想充分利用多核，但运行顺序却得不到保证，真让人伤脑筋。

再次挑战多核化

于是我开始思考能够保证运行顺序的结构。我想到了几种方法，其中最省事的方法就是将某个任务固定由某个工作线程运行。只要一个任务由同一个工作线程运行，就会按照进入队列的顺序依次处理。

因此我在第 1 版的基础上进行了以下修改。

- 每个线程都拥有一个任务队列
- 首次运行时决定运行任务的线程

运行任务的线程是固定的，这样就不会出现顺序改变的情况，程序也就可以正常工作了。

此外，我还制定了下面的规则，以进一步有效利用多核 CPU。

第 1 个规则是在最开始启动像文件输入这样的生产者任务时，把它添加到等待处理的任务最少的队列中。这么做的目的是分散工作线程的负荷。第 2 个规则是在工作线程 n 中运行的任务进行 emit 时，运行进行 emit 的任务的工作线程（如果还没有确定线程的话）必须为工作线程 n+1。这样一来，当存在多个管道时，就极有可能无须等待第一个任务结束就可以开始下面的任务。

第 2 版的程序结构如图 2-40 所示。

任务队列中的任务基本上是按照最初投入的顺序依次运行的。工作线程从队列中依次取出任务运行并进行循环，在管道全部结束之后，循环也随之结束。

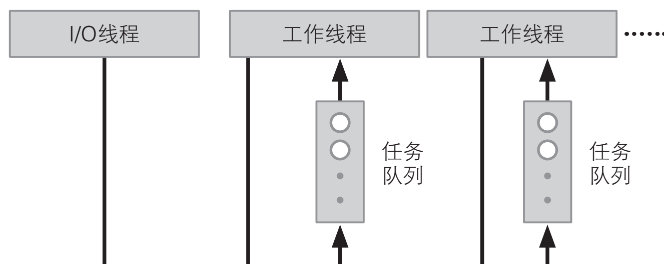


图 2-40　多线程版本（第 2 版）

带优先级的队列

通过前面的改善，事件循环的多线程化取得了成功，不过有一点让我有些在意。

管道中的任务是生产者→过滤器→……→消费者这种形式，生产者开始管道处理，数据被后续的过滤器依次加工之后，最后由消费者消费掉。一旦管道段数变多，管道中越靠后的任务运行速度就可能越慢。

为了方便大家理解，我们来看一个例子。在这个例子中，CPU 为双核，管道有 3 段。在后面的讲解中，我们把第 *n* 个工作线程称为“工作线程 *n*”，第 *n* 个队列称为“队列 *n*”，第 *n* 段管道的任务称为“任务 *n*”。

在这个例子中，处理按如下步骤进行。

(1) 任务 1（生产者）在工作线程 1 中进行 `emit`，在队列 2 中添加任务 2

(2) 任务 2（过滤器）在工作线程 2 中进行 `emit`，在队列 1 中添加任务 3

(3) 任务 3（消费者）在工作线程 1 中运行

不要忘记，在工作线程 1 中进行 `emit` 之后，任务 1 也会把自己添加到队列 1 中，而且有可能先于上述第 2 步的在队列 1 中添加任务 3 这一处理完成。这样的话，在一个数据在管道中流转完之前，另一个数据就已经被投入到管道中了。这就意味着数据一个接一个地被生产出来，任务堆积在队列中，但工作线程的处理速度却赶不上数据生产的速度。这样下去队列会越来越长，内存就会被浪费，因此我们需要让生产和消费保持平衡。

为了解决这个问题，这里使用了带优先级的队列。也就是说，相比在管道前方的生产者任务，优先处理加工数据的过滤器任务和消费者任务。这个方法的作用在于防止队列过长。

于是我稍微修改了一下队列的实现。

队列的实现

下面我们就来看一下队列的实现。图 2-41 是表示 Streem 队列的结构体。

```
struct strm_queue_task {
  strm_stream *strm;
  strm_func func;
  strm_value data;
  struct strm_queue_task *next;
};

struct strm_queue {
  pthread_mutex_t mutex;
  pthread_cond_t cond;
  struct strm_queue_task *fi, *hi, *fo;
};
```

图 2-41　Streem 的队列结构体

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

这个结构体的本质是 `strm_queue_task` 结构体的链表。队列中任务的添加和取出处理如图 2-42 所示，不过这里只摘录了最基本的部分。

```
● 添加
void
strm_queue_push(strm_queue *q,
        struct strm_queue_task *t)
{
  if (q->fi) {
    q->fi->next = t;
  }
  q->fi = t;
  if (!q->fo) {
    q->fo = t;
  }
}

● 取出（运行）
int
strm_queue_exec(strm_queue *q)
{
  struct strm_queue_task *t;
  strm_stream *strm;
  strm_func func;
  strm_value data;

  t = q->fo;
  q->fo = t->next;
  if (!q->fo) {
    q->fi = NULL;
  }

  strm = t->strm;
  func = t->func;
  data = t->data;
  free(t);

  (*func)(strm, data);
  return 1;
}
```

图 2-42　队列中任务的添加和取出处理（摘录）

队列的基本结构是从 `strm_queue` 结构体的 `fo`（first out）成员变量开始连在一起的链表（图 2-43）。取出任务时，从 `fo` 取出一个任务，然后将 `fo` 修改为指向下一个任务。添加任务时，使链表末尾的任务的 `next` 指向新的任务。通过遍历链表寻找末尾的做法效率很低，因此就让 `fi`（first in）成员变量指向末尾。

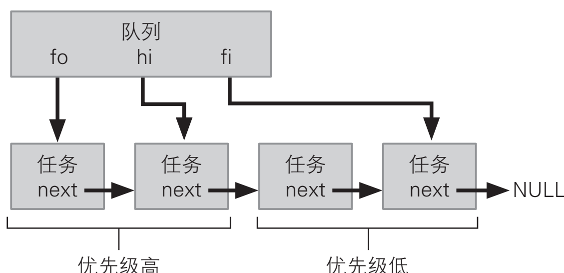


图 2-43　使用链表的队列

优先级的实现

接下来实现队列的优先级，做法是在 `fi` 和 `fo` 之间增加 `hi`（high priority input）指针。在添加优先级低的（生产者）任务时，和之前一样在 `fi` 处添加任务即可。

除此之外的任务都需要添加到 `hi` 指向的“高优先级任务的末尾”，这样从 `fo` 开始连在一起的整个链表的任务就会按照优先级从高到低的顺序排列。从队列中取出任务时，不用考虑优先级，直接从前面取出即可。

并发控制

向任务队列中添加任务的线程和取出任务并运行的线程在大部分情况下是不同的线程。

这就意味着在某个线程正在修改队列时，如果别的线程也要修改队列，就可能会破坏链表的完整性。

为了规避这个风险，在有可能发生多个线程同时修改数据的情况时，就需要进行并发控制，以使实际的修改分别独立进行，这时就用到了 mutex。图 2-41 的 strm_queue_task 结构体中就增加了 pthread_mutex_t 型的 mutex 成员变量。

```
(1)初始化
pthread_mutex_init(&mutex, NULL);

(2)lock
pthread_mutex_lock(&mutex);

(3)unlock
pthread_mutex_unlock(&q->mutex);
```

图 2-44　pthread_mutex_t 的使用方法

mutex 的使用方法很简单（图 2-44）。首先进行初始化，然后将需要进行并发控制的“危险区域”用 lock 和 unlock 包围起来。

在队列的实现中，修改 q->fi、q->hi、q->fo 的部分是“危险区域”，需要用 lock 和 unlock 围起来。这样一来，用 lock 和 unlock 围起来的部分就会被限制为只能有一个线程运行。

需要注意的是，mutex 只能进行显式的并发控制。假如忘记使用 lock 和 unlock 围住修改数据的处理，就会出现棘手的 bug。

队列为空时的处理

还有一点让我比较在意，就是当队列为空时应该如何处理。当队列中没有要运行的任务时是没有什么事情可做的，这时就需要等待任务被添加到队列中。

最简单的实现方法是一边循环一边频繁地查看队列，检查是否有任务到来，这种方法被称为“忙循环”（busy loop）。不过，到任务到来为止忙循环一直会处于运转状态，这就白白浪费了 CPU 的计算资源，因此这种做法只有在确定等待时间较短的情形下才能使用。

这次使用的是 POSIX thread 库的“条件参数”功能。条件参数与互斥锁 mutex 组合使用。例如图 2-45 的程序的 (a) 部分，在 mutex 锁定期间调用 pthread_cond_wait() 会使当前线程进入等待状态，直到被其他线程唤醒。

```
(a)使用条件参数时的等待方法
  pthread_mutex_lock(&mutex);
  while (!fo) {  // fo可能还没有被设置
    pthread_cond_wait(&cond, &mutex);
  }
  pthread_mutex_unlock(&mutex);

(b)使用条件参数时的唤醒方法
  pthread_mutex_lock(&mutex);
```

```
// 满足条件时的处理
// 满足条件时“唤醒”
pthread_cond_signal(&cond, &mutex);
pthread_mutex_unlock(&mutex);

// 在有多个线程等待的情况下，用signal唤醒其中一个
// pthread_cond_broadcast用于一次唤醒全部线程
```

图 2-45　条件参数的使用方法

在运行队列中的任务时，如果队列为空就调用 `pthread_cond_wait()`，并一直保持等待状态，直到有任务被追加到队列中。图 2-46 展示了并发控制和条件变量组合使用的版本，请大家看一下它与图 2-42 有什么不同（图 2-46 中也增加了对 `q->hi` 的处理）。

```
int
strm_queue_exec(strm_queue *q)
{
  struct strm_queue_task *t;
  strm_stream *strm;
  strm_func func;
  strm_value data;

  pthread_mutex_lock(&q->mutex);
  while (!q->fo) {
    pthread_cond_wait(&q->cond, &q->mutex);
  }
  t = q->fo;
  q->fo = t->next;
  if (t == q->hi) {
    q->hi = NULL;
  }
  if (!q->fo) {
    q->fi = NULL;
  }
  pthread_mutex_unlock(&q->mutex);

  strm = t->strm;
  func = t->func;
  data = t->data;
  free(t);
```

```
  (*func)(strm, data);
  return 1;
}
```

图 2-46　从队列中取出任务运行

图 2-46 的关键点在于围在 pthread_cond_wait() 外面的条件部分的语句不是 if 而是 while。其实最开始是用 if 来检查的，但是有人指出 pthread_cond_wait() 在条件不成立的情况下也可能会因为某些原因而直接返回，所以需要用 while 来检查。这不禁让我觉得并发编程真是深奥。

添加任务时也和这段代码一样，用 mutex 的 lock 和 unlock 围住操作队列的数据的部分，在末尾调用 pthread_cond_signal()。

多线程处理的调试

像这样，我每次都根据连载的内容去考虑开发 Streem 语言处理器的哪一部分，出于一种必须把连载写完的使命感，开发进度虽然缓慢但还是在稳步进行。

这次讲解的部分我在写原稿时也进行过调试。当然我是在确认程序运行没有问题之后才写的原稿，但是完美的程序不是一次就能写出来的，我还是进行了多次调试。

其中，多线程程序的调试尤为麻烦。经过这次修改，Streem 的语言处理器变为多线程版本了，这就使调试变得愈发麻烦。

在调试单线程程序时会使用 gdb 等调试器，但是我觉得 gdb 在调试线程时并没有那么好用。可能是我孤陋寡闻，如果有读者精通使用调试器调试线程，还望不吝赐教。

那么我是如何调试的呢？说起来不怕人笑话，我是用原始的 printf 方法调试的。使用这种方法时需要注意以下几点。

第 1 点是调试的输出应输出到 stderr（标准错误输出），这样程序本身的输出与调试的输出就可以区分开来。比如下面这条命令能够统计标准输出的行数、字数和单词数，而调试的输出会显示在控制台（console）上。

```
% a.out | wc
```

如果想让调试输出的内容也一起在屏幕上滚动，可以使用下面这条命令把标准输出和标准错误输出的内容混合在一起显示。

```
% a.out |& lv
```

第 2 点是需要有 `fprintf` 这样的功能来输出想在调试器中查看的值。比如，当你想检查结构体的成员是否被初始化，或者是否按照正确的顺序运行了处理时，使用 `fprintf` 的 `%p` 格式化输出符就很方便。`%p` 是用于显示指针地址的格式化输出符。普通的地址用十六进制显示，`NULL` 的值用“`(nil)`”显示，非常好用。

第 3 点是提交代码时用 `git diff` 等命令检查是否已全部删除用于调试输出的 `fprintf` 语句。把带着调试输出的代码提交到互联网上，那可有点丢人。

用结构体表示对象

到目前为止，Streem 的数据都是将结构体和字符串转换（cast）为 `void*` 进行传递的。我们需要注意类型是否一致，如果弄错了程序就会崩溃。

但是作为一个语言处理器还是应该把对象实际的类型信息管理起来。Streem 中使用图 2-47 那样的结构体来表示对象。

```
enum strm_value_type {
  STRM_VALUE_BOOL,
  STRM_VALUE_INT,
  STRM_VALUE_FLT,
  STRM_VALUE_PTR,
};

typedef struct strm_value {
  enum strm_value_type type;
  union {
    long i;
    void *p;
    double f;
  } val;
} strm_value;
```

图 2-47　Streem 中对象的表示方法

Streem 中表示对象的 `strm_value` 是一个简单的结构体，由表示数据的 `union`（联合体）和表示类型的 `type` 成员构成。C 的数据与 `strm_value` 之间的相互转化使用图 2-48 中的函数。我会在 2-6 节之后详细介绍这些函数的使用方法。

```
strm_value strm_ptr_value(void*);
strm_value strm_bool_value(int);
strm_value strm_int_value(long);
strm_value strm_flt_value(double);

#define strm_null_value() strm_ptr_value(NULL)

void *strm_value_ptr(strm_value);
long strm_value_int(strm_value);
int strm_value_bool(strm_value);
double strm_value_flt(strm_value);
```

图 2-48　strm_value 转换函数

除了这次采用的结构体的方式之外，表示对象的方法还有 Python 使用的包括数值在内的对象全部用指针表示的方式、LuaJIT 使用的将类型信息用浮点数表示的 NaN Boxing 方式、Ruby 使用的把整数插入到指针里的“带标签的指针”方式等。这些技术都非常有趣，今后我会详细介绍。

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

GC

可以创建对象之后，就必须能够回收不再使用的对象，也就是进行 GC（Garbage Collection，垃圾回收）。

GC 的实现方法有好几种，这次我们使用其中最简单的 libgc 方法来实现。

libgc 的正式名称是 Boehm-Demers-Weiser's GC，是一个垃圾回收库。原则上 C 和 C++ 程序中只需链接这个像是有魔法一样的库，就可以在 malloc 分配的内存空间不再使用时自动进行回收。这个库也支持多线程，因此可以用在 Streem 语言上。

在使用 libgc 之前，首先需要安装这个库。我们使用 apt-get 等包管理器安装 libgc 包。在 Debian 系列 Linux（Ubuntu 等）中包名为 libgc-dev，在 Red Hat 系列（Fedora 等）中包名为 libgc-devel。

```
% apt-get install libgc-dev
```

安装之后，在程序代码的前面调用下面的函数，并根据表 2-2 修改程序。

```
GC_INIT();
```

这里需要补充说明一下 calloc() 和 free() 的相关内容。calloc() 这个函数用于把分配的内存空间清零，在 libgc 中没有对应的函数。但是 GC_MALLOC() 能够保证将分配的内存空间清零，所以我们只需调整参数即可。在进行垃圾回收的 libgc 中，free() 本来就是不需要的，所以基本上会删除 free() 的调用语句。

表 2-2　使用 libgc 时的修改方法

原来的函数	使用 libgc 的函数
malloc(size)	GC_MALLOC(size)
realloc(p,size)	GC_REALLOC(p,size)
calloc(n,size)	GC_MALLOC(n*size)
free(p)	GC_FREE(p) 或者删除

但是在出于某些原因需要强制释放内存时，就需要使用 GC_FREE()。在这种情况下，使用者需要注意不要不小心释放了仍在使用的内存空间。

程序修改完之后，打开 Makefile 增加链接 libgc 的规则就可以了。很简单吧。

当前我们使用 libgc 进行垃圾回收，不过 libgc 并不是万能的，也存在一定的局限性。比如，在使用了修改指针的值的技巧时就不能正确地进行 GC，另外，因为内部利用了汇编语言，所以很难支持所有平台。

Streem 对象不可变的特点有利于实现引用计数方式的 GC 和分代 GC。好好利用这一点，也许能开发出比通用的 libgc 更为高效的 GC。我们把这个课题留到将来去研究。

被“搁置”的语法分析也取得了进展

最近我一直忙于事件循环的开发，于是语法分析部分就被搁置了下来。

但这并不意味着语法分析没有取得任何进展，在 Streem 开发初期用 Go 语言独自开发了语言处理器（Streeem）的 Mattn（松本康弘）先生等很多人都向我发送了 Pull Request，得益于此，Streem 实现了以下功能。

- 语法树的生成
- 简单的解释器

没有我的主导，也没有密切的交流，在这样的情况下开发还能不断地向前推进，这着实让我感到惊讶。Streem 的开发屡屡刷新了我对开源的认识。

小结

Streem 的核心实现也支持了多线程，语言的完成度越来越高，但还是没有达到实用程度。接下来就需要把语法分析部分和核心部分组合到一起，使其成为可以使用的语言。敬请期待。

时光机专栏

GC程序中有错误

本节是 2015 年 4 月刊中刊登的内容，实现了通过事件循环进行任务处理的多线程版本。这次是通过 `mutex` 的带优先级的队列来实现的。实际上这部分内容后来因引入了无锁算法等而被完全替换掉了，但为了记录当时的过程，成书时我还是选择保留了下来。

现在重读原稿让我想起了调试时的艰辛，我还是忍不住发牢骚："多线程编程的调试真是太痛苦了！"因为 bug 的发生与运行时机有关，所以很难查明问题是在什么样的状态下发生的。

为了了解程序的状态，我尝试了调试器，或者在程序中插入 `printf` 语句，这样一来运行时机不一样了，bug 又不出现了。这种事情经常发生，让人欲哭无泪。虽然之后我对事件循环做了很多改进，但老实说，我现在也不敢保证 Streem 的事件循环里没有 bug。

关于 `libgc` 的使用，这里也有必要再补充说明一下。连载时没有表 2-2 那种关于程序修改的说明。我记得旧版本的 `libgc` 替换了 `malloc()` 等函数的实现，只要链接 `libgc` 就能进行垃圾回收，所以没有仔细经过验证就落笔了。

示例程序可以运行，所以我没有注意到其实它并没有进行垃圾回收，而是做了内存的分配，完全没有释放这些内存。只是因为示例程序中处理的数据不是很多，所以才没有出现问题。真是太不好意思了。

2-6 缓存与符号

本节将通过Streem的实现来介绍内存访问的详细内容。在多核环境中，有效利用缓存是特别重要的。另外，我们还将通过语言的设计来介绍符号（symbol）这一数据类型。

在 2-5 节中，为了有效利用多核，我让 Streem 支持了多线程，但我发现了一个问题。在 2-5 节的实现中，当管道中有任务排队时，为了最大限度地利用多核而把各个任务分配到了其他线程去执行，但是仔细想想，这并不是一种很妥当的做法。因为我发现，虽然在任务数量不多时没什么影响，但是当任务数量较多时，某个意想不到的地方可能就会给运行性能带来影响。为了能够使 Streem 在今后不断改善，这次我们先来考察一下这个问题。

■ 有效利用缓存

这里我们要讨论的是内存访问速度的问题。我们平时在写代码时，基本上不会在意内存访问花了多少时间，甚至也不会在意是否给变量分配了内存、是否分配了 CPU 寄存器等，但是对 CPU 来说，从不同的地方取出数据的差别是非常大的。

CPU 只需用 1 个时钟周期即可从寄存器中取出数据。2 GHz 的 CPU 的 1 个时钟周期是 0.5 ns。可同样的数据如果是在主存上，就需要访问外部总线，从速度慢但容量大的内存传送数据，因此花费的时间要多得多（几百倍）。

在这期间，CPU 拿不到需要的数据只能等待。这就意味着每次从主存取出数据时，CPU 只能发挥出几百分之一的性能。

用缓存高速访问数据

这样一来就白白浪费了 CPU 的性能，所以很久之前 CPU 就设置了缓存来解决这个问题。

“缓存”的英语是“cache”，和现金“cash”稍有不同。据说“cache”这个词原本在法语中是“隐秘的场所”“储藏室”的意思，后来意思发生了改变，在计算机领域表示“临时保存取出的数据（或者频繁访问的数据）的存储空间”。

具体来说，CPU 中设置了与其直接相连的、容量虽小但速度很快的存储空间，从内存中取出的数据也会保存在这个缓存空间里。当再一次访问同一个地址的数据时，就会直接使用缓存中的数

据，而不用去访问速度缓慢的主存，这样就可以避免访问主存所引起的性能低下的问题。

设置多级缓存

不过我们无法配置较大容量的高速缓存。现在我手头的计算机（CPU：Intel Core i7 2620M）仅搭载了两个容量为 64 KB 的缓存，一个用于数据，一个用于命令。这个容量在 2015 年最新的 CPU 上也没有发生改变。

于是，为了尽可能地减少对缓存上没有的数据的访问，现在的 CPU 都搭载了多级缓存（图 2-49）。比如 Core i7 搭载了 64 KB 的一级（L1）缓存，还搭载了没有 L1 那么快但容量更大的 512 KB 的二级（L2）缓存，以及容量进一步增大到 4 MB 的三级（L3）缓存。

很多程序都有频繁访问同一地址的数据的特点，有了 4 MB 的缓存，很可能不用访问主存就能完成处理。

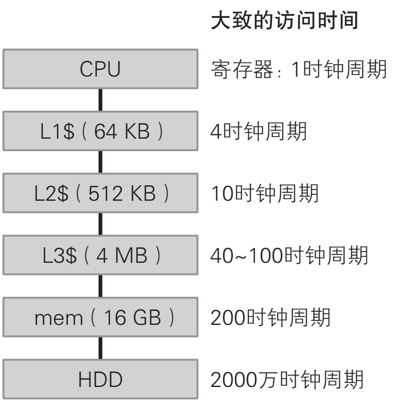


图 2-49　多级缓存

本图中的时钟周期只是估算而已。访问时间在每个 CPU 上都不同，而且还会受时钟周期之外的因素影响，因此很难表示出精确的数值。“L1$”中的“$”是“缓存”的缩写，它出自于和 cache（缓存）发音相同的 cash（现金）一词

数据更新时的问题

在多核环境下，通常每个内核都拥有独立的 L1 缓存，然后多个内核之间共享 L2 和 L3 缓存。

缓存只是临时保存数据的场所，因此在更新数据时，为了之后也能读取到最新数据，我们需要把数据写回到主存。另外，如果其他内核更新了内存上的数据，那么当前缓存上的数据就会成为无意义的旧数据。这时就需要让旧的缓存失效，然后重新读取数据。

这其实是一个非常复杂的问题，菲尔·卡尔顿（Phil Karlton）就说过：“计算机科学有两大难题，分别是缓存失效和命名。”这里不打算深入探讨这个难题，我只是想告诉大家，看上去很简单的内存访问，其实在 CPU 内部进行着非常复杂的处理。

对管道的再次思考

让我们回到原来的话题。本节开头提到过，当管道中有任务排队时，为了最大限度地利用多核，会把各个任务分配到其他线程执行。这样一来，流转在管道中的数据就会被不同的线程访问，从而减少了有效利用高速的 L1 缓存的机会。

也就是说，要想有效利用限制极其苛刻的缓存容量，就需要尽量从同一个内核去访问同一个数据。如果从多个内核访问同一个数据，就会在宝贵的 L1 缓存中保存重复的数据，先前访问的数据就会被移出缓存。这实在是太浪费资源了。

这种现象在多个管道工作时会更加明显。现在运行在 Streem 中的还是非常简单的管道，没有太多浪费缓存的情况，如果管道变多，这个问题就会逐渐显露出来。

在同一个内核中运行速度更快

我们来看一下图 2-50(1) 这个管道。最开始的任务是从标准输入读取 1 行字符串，读取的字符串数据经过 L1 → L2 → L3 缓存，最终被写入主存（图 2-50(2)）。

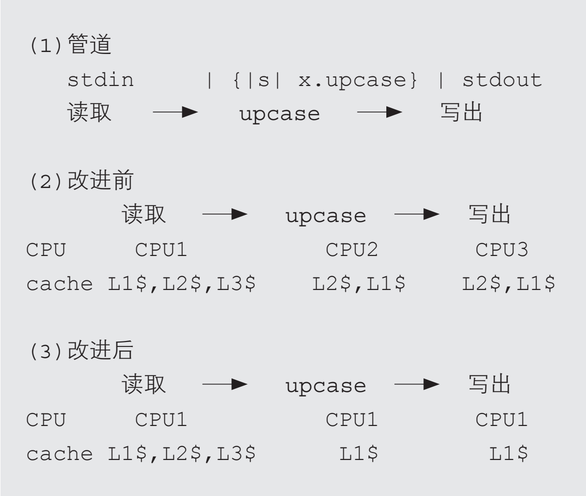


图 2-50　管道上的任务和缓存

将字符串转换为大写字母的下一个任务如果在其他线程执行（也就是说很可能会在其他内核执行），那么数据就不会存在于这个内核的 L1 缓存中，所以只好去 L2 缓存中寻找数据。在大多数情况下，L2 缓存会在内核之间共享，所以数据还有可能残留在 L2 缓存中，这时从 L2 缓存中取出数据就行了。但是如果有其他管道在同时运行，那么 L2 缓存或许也会被数据占满，这时就要到 L3 缓存去找，如果 L3 缓存也满了，就需要去访问主存。

但是，在尽量把同一个管道的任务分配给同一个内核执行的情况下，对缓存的访问就会按照图 2-50(3) 进行。如果任务都在同一个内核执行，那么数据就很可能保存在 L1 缓存中，只要缓存未满，也许只访问高速的 L1 缓存就能完成处理。

要意识到缓存的存在

现代 CPU 上运行的软件的性能与缓存的使用有着很大的关系。对软件来说，对缓存的访问是“透明的”，除了性能之外，很难在其他方面感受到缓存的存在。但是对性能来说，情况却大不相同。如果没能有效使用缓存，访问速度可能就会慢上几十倍。在开发 21 世纪高速运行的软件时，我们不能忽视缓存的作用。

可是，灵活使用肉眼看不见的缓存是非常困难的。特别是在与线程纠缠在一起的情况下，要理解缓存的状态简直就超过了人类的理解范围。即使依靠推测去修改代码，也很难使性能得到改善。

这时需要做的是测量。正确的测量才是改善性能最好的办法。Linux 上测量缓存活动的工具有 oprofile 和 cachegrind 等。这些工具不仅可以测量某个函数花费了多少时间，还可以报告各个命令的缓存缺失所导致的延迟等。关于这些工具的使用方法，我会在将来讲解 Streem 的性能改善时一并进行说明。

■ 符号的处理

接下来是有关语法的内容。

Ruby 中有符号这种数据类型，用于表示变量名和标识符，有自己的名字。从这个角度来看，符号与字符串很相似，不过也有几点不同，如下所示。

- 同一个名字的符号只能有一个
- 能够快速判断是否一致（不需要检查内容）
- 无法像 Ruby 的字符串那样修改内容

Ruby 利用了符号速度快的特点，在指定方法名、变量名以及关键字参数等场景中广泛使用了符号。

Ruby 的符号是从 Lisp 继承过来的。在开发 Ruby 之前，我便受到了 Lisp 很大的影响，所以引入符号也是顺理成章的事情。

Lisp 的符号

在 1958 年诞生的 Lisp 中，符号与列表都是 Lisp 基本数据类型的一种。据说最早版本的 Lisp 中就已经有符号了，那时反而还没有字符串，现在我们使用的字符串在当时都是用符号代替的，所以符号是比字符串更古老的一种数据类型。

在 Lisp 中，符号被用在了各种各样的场景中。Lisp 的程序本身虽然可以用列表结构来表示，但其中出现的变量名和函数名等与名称有关的内容都是用符号表示的（图 2-51），可见符号在 Lisp 中的重要性。

```
; 使用Lisp开发的阶乘程序
(defun fact (n)
  (if (eq n 1)
      1
    (* n (fact (- n 1)))))

; 有底纹的部分是符号
```

图 2-51　Lisp 程序中的符号

初学者的困惑

于是 Ruby 也引入了符号，但是一些不了解 Lisp 的 Ruby 学习者表示难以理解字符串和符号为何不同。

对我来说符号和字符串不同是理所当然的事情，所以一开始我不能理解为何有人对此表示不满。当我多次听过他们的意见之后，就渐渐地了解了他们不满的原因。

也就是说，先不考虑内部数据的结构，从表面来看，字符串和符号都是“表示字符序列的内容”。从这个角度来看，符号只不过是不能修改的字符串，但支持的操作（方法）却少了很多，这让人感觉很不方便。我收到了很多“用起来不方便，请添加和字符串同样的方法吧”“统一字符串和符号吧”之类的请求，那时我才明白，原来大家对字符串和符号的理解与我完全不同。

之所以会发生这样的“误解”或者说“认知分歧”，是因为在没有像 Ruby 那样受到 Lisp 强烈影响的语言中，是没有出现符号这种概念的，但这并不是说这些语言的内部就不需要符号这样的东西。

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

其他语言中相当于符号的概念

那么，那些没有符号的语言是如何实现符号的功能的呢？

最简单的方法是用普通的字符串代替所有符号。如果不在意性能，也可以使用普通的字符串进行内部标识符（名称）的管理，操作也没有那么困难。

使用普通的字符串作为标识符的最大缺点是，在进行比较时，花费的时间与字符串的长度成正比。要判断字符串 `"abcd"` 和 `"abce"` 的不同，就需要依次检查每个字符，直到第 4 个字符。如果用普通的字符串实现像标识符这种需要频繁去比较的东西，就很容易出现性能问题。

于是 Python 等语言便没有引入符号的概念，而是在字符串上下了一番功夫，以此来避免性能问题。

具体来说，满足一定条件的字符串在拥有同样内容时返回同一个对象（intern 化），这样就可以在不看内容的情况下进行一致性检查了。

所谓“一定条件”，在 Python 中是指以下两个条件中的任意一个。

- 一定长度（默认是 20 字符）以下的字符串字面量
- 显式声明了 `intern` 的字符串

而 Lua 语言则是在原则上对所有字符串进行符号化，也就是说，内容相同的字符串全部是同一个对象。

Python 和 Lua 之所以能做到这一点，是因为 Python 和 Lua 中字符串是不变的，也就是说，内容是不能修改的。像 Ruby 这样的可以修改字符串内容的语言是不能这么做的。真是让人难以选择。

Streem 中实现符号功能的方法

我们回到 Streem 的话题。Streem 这样的语言也需要符号这种概念。如何实现这个功能是语言设计上的一项重要决策。

是像 Ruby 那样引入独立的符号数据类型，还是像 Python 和 Lua 那样在使用字符串的基础上设法发挥符号的作用呢？

实际上 Streem 有一个设计原则，就是如果不是特别想用，并且也没有什么明显的技术优势，就选择与 Ruby 不同的做法。Streem 的对象原则上不可修改、代码块结构中使用“{ }”等都是这个设计原则的体现。

基于这个原则，我打算在符号的设计上采用与 Ruby 不同的做法。幸运的是 Streem 中的字符串是不可修改的，所以我决定让它和 Python 一样使用字符串来发挥符号的作用。

修改字符串生成函数

具体的做法是修改 Streem 的字符串生成函数，让有同样内容的字符串成为同一个字符串。

新的字符串生成步骤如下所示。

1. 准备一个保存字符串对象的散列表。散列表的键是指针（const char*）和长度（size_t），值是字符串对象（struct strm_string*）
2. 生成字符串时首先根据传来的数据（指针和长度）查找散列表，如果找到了对象，就返回这个字符串对象
3. 如果没找到对象，则生成字符串对象，并保存到散列表中

前面的处理都不是很难，实际按照这个步骤编写的生成字符串对象的函数 strm_str_new() 如图 2-52 所示。

```
#include "khash.h"

/* 散列表的定义 */
KHASH_INIT(sym, struct sym_key,
  struct strm_string*, 1, sym_hash, sym_eq);

/* 保存符号的散列表 */
static khash_t(sym) *sym_table;

/* 字符串对象的分配 */
static struct strm_string*
strm_str_alloc(const char *p, size_t len)
{
  struct strm_string *str
    = malloc(sizeof(struct strm_string));

  str->ptr = p;
  str->len = len;
  str->type = STRM_OBJ_STRING;

  return str;
}

/* 字符串对象的生成函数 */
struct strm_string*
strm_str_new(const char *p, size_t len)
{
  khiter_t k;
  struct sym_key key;
  int ret;

  /* 符号表的初始化 */
  if (!sym_table) {
    sym_table = kh_init(sym);
  }
  /* 是否存在于符号表中 */
  key.ptr = p;
  key.len = len;
  k = kh_put(sym, sym_table, key, &ret);
  /* 存在: ret == 0 */
  /* 不存在: 可以插入到k的位置 */

  if (ret == 0) {               /* found */
    /* 如果找到了对象则返回回该对象 */
    return kh_value(sym_table, k);
  }
  else {
    /* 如果没有找到则分配对象 */
    struct strm_string *str;

    /* allocate strm_string */
    if (readonly_data_p(p)) {
      /* 如果是只读空间则不需要复制 */
      str = strm_str_alloc(p, len);
    }
    else {
      /* 复制字符串数据 */
      char *buf = malloc(len);
      if (p) {
        memcpy(buf, p, len);
      }
      else {
        memset(buf, 0, len);
      }
      str = strm_str_alloc(buf, len);
    }
    /* 将生成的对象添加到符号表 */
    kh_value(sym_table, k) = str;
    return str;
  }
}
```

图 2-52　支持符号特性的 strm_str_new()

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

需要注意的是，这里使用了 `khash` 库来实现散列表。使用 `khash` 库时，只需包含头文件就可以使用散列表。我们也可以使用宏来实现模版类型这种功能，可以把任意类型当作键和值。

线程问题

这样一来，Streem 就可以和 Lua 一样使用字符串来发挥符号的作用了，而且系统会保证内容相同的字符串是同一个对象。

但其实这个实现还不完整，至少还存在两个问题。

首先是支持多线程的问题。图 2-52 的程序中包含查看和添加符号表数据的代码，如果多个线程同时访问符号表，在最坏的情况下数据会遭到破坏，因此我们需要使用并发控制等方法支持多线程访问。

我想到了两个解决办法。一个办法是用 `mutex` 围住访问符号表的代码（具体指 `kh_put` 的调用）。虽然 `lock/unlock` 操作每次至少耗费 100 ns，但还算在误差范围内。这种做法只需添加几行代码即可，可以快速解决这个问题。

另一个办法是只在事件循环开始之前，也就是多线程还没有运行的时候向符号表添加数据，在多线程运行时不向符号表添加数据。这种做法由于不需要进行并发控制，所以在多线程环境下不会对性能造成影响（或者说影响较小）。

实际上，会成为符号的字符串一般是在程序的初始化阶段创建的，所以我感觉这种做法可行，只是在实现上有些复杂，而且即使是同一个字符串，也存在会添加到符号表和不添加到符号表两种情况，这又需要增加相应的处理。

比较下来还是第一个办法占据绝对优势，所以这次我决定采用第一个办法。不过考虑到接下来要解决的问题，可能还需要在将来探讨其他办法，特别是后一个办法。

符号垃圾问题

第二个问题是“符号垃圾问题”。每次处理字符串时，所有字符串都会被添加到符号表中。如果有大量内容不同的字符串出现，它们就都会被添加到符号表中，这就可能导致内存不足。

Ruby 的字符串与符号分离，理论上不容易发生这种问题，但也被发现了漏洞，那就是允许攻击者通过外部输入生成符号，从而大量消耗内存，妨碍程序运行。因此，在 2014 年 12 月发布的 Ruby 2.2 版本中，符号也成为了 GC 的对象。在 Ruby 2.2 以后的版本中，不使用的符号会被自动回收。

当然，Streem 也有发生同样问题的风险，将来也需要用某种方法解决。

不过这个问题不需要马上解决，这里我们只是探讨一下而已，具体的实现则当成今后的课题。

符号的 GC

那么将来在解决符号垃圾问题时，有哪些方法可供参考呢？

其中一个方法是像 Ruby 一样把符号当成 GC 的对象。在这种情况下，将符号表中对字符串的引用作为弱引用（虽然是引用，但不会成为 GC 保护对象），在字符串被回收时从符号表中去除。

另一个方法是像 Python 一样只把符合条件的字符串添加到符号表，以避免符号表过大。前面提到过的把字符串分为添加到符号表和不添加到符号表就是指这种方法。

但这种方法有几个缺点。首先，存在不添加到符号表的字符串就意味着可能需要频繁比较字符串的内容，这就失去了符号自身的优点。

再者，光靠这种做法无法完全避免符号表过大的问题。实际上，Python 也会把已经添加的字符串当成 GC 的对象。

考虑到这些，就可以明白将来解决符号垃圾问题时需要实现符号的 GC。

不过现在 Streem 的内存管理使用了 `libgc`，所以不需要担心动态分配（`malloc`）的内存空间的释放问题，但是要实现前面提到的弱引用等就很困难了。

要实现符号的 GC，就需要进行粒度更细的控制，因此就需要放弃 `libgc`，去实现自己的 GC。

小结

本节探讨了两个主题：缓存与内存访问的成本，以及编程语言中符号的设计。

让我觉得有趣的是，像内存访问这种稀松平常的事情，其背后也有平时看不到的缓存机制在支撑。另外，缓存可能会使软件性能发生几倍甚至几十倍的变化，这一点也让我觉得非常有意思。

就连编程语言中的标识符（名称）的处理等看起来非常琐碎的内容，也需要经过多方面的考虑。这也是语言设计上不太为人所知的有意思的地方。

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

时光机专栏

不同年龄段的人的认知偏差

本节是 2015 年 5 月刊中刊登的内容。现代计算机的复杂程度远远超出了我们的想象，缓存带来的运行效果的变化就是一个例子。

在我们使用的编程模型中，只能把数据看作二次存储空间中的文件，或者一次存储空间的堆或变量，但实际上变量有的被分配到了寄存器中，有的被分配到了内存中，表面上看起来完全相同，但访问时间却相差甚远。另外，在访问被分配到内存的变量和堆时，是否使用缓存也会给访问时间带来极大的影响。开发时能否注意到这一点将会大大影响软件的性能。

话虽如此，但现在 Streem 的实现还没有把效率纳入考量，Streem 还没到使用缓存来提高性能的阶段，所以大家把本节内容当成软件性能的相关读物更为合适。

针对符号这个主题，我这里再补充几句。正文中也写到了，对于受到 Lisp 强烈影响的我来说，符号和字符串不同是理所当然的，我对此从来没有质疑过，但是不太了解 Lisp 的“年轻”一代却觉得二者的区别没有任何意义，这让我觉得很有意思。各自背景的不同导致了认知差异，这次的小插曲就是这种认知差异的一个典型例子。随着时代和背景的变化，常识也会发生变化，我切身感受到了这一点。

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

2-7 转换为抽象语法树

本节将修改Streem的语法分析器，无论如何也要让Streem作为一门语言运行起来。我将把语法分析的结果转换为抽象语法树，虽然进一步转换为虚拟机的机器码会提高程序执行效率，但是这里我还是选择先让语言运行起来。

虽然 Streem 的内部结构一点点地实现了，但它还是不能作为一个语言运行起来，所以这次我想修改 Streem 的语法分析器，无论如何也要让它达到能运行的程度。

在这之前我要先改进一下 2-6 节的实现。在 2-6 节，为了支持符号语法，我把所有的字符串都添加到了符号表中，让内容相同的字符串成为同一个对象。

但在经过多次测试后，我发现这种做法不太好。因为把所有字符串都添加到符号表之后，随着读入数据的增多，内存的使用量也会变得越来越大。

在 2-6 节提到过，Lua 也采用了同样的做法，所以我觉得应该没什么问题，不过现在看来还是亲自验证一下比较靠谱。

支持符号语法的新方法

于是我决定按照以下方式来实现。

首先，在进入事件循环之前，也就是在单线程运行时，由于不用担心并发控制的问题，所以在生成字符串时将字符串添加到符号表。不过这里参考了 Python 的做法，不将超过一定长度（暂定为 64 个字符以上）的字符串添加到符号表。

当事件循环开始，进入多线程模式后，为了避免竞争，Streem 在生成字符串时不去访问符号表，而是每次生成一个新的字符串。在这种情况下，即使字符串内容相同，对象也不同。

但有时出于某些原因还是需要用到符号化的字符串，因此我开发了新的函数 `strm_str_intern()` 来获取符号化的字符串。这个函数在被调用时会并发地访问符号表，返回符号化的字符串。

经过这一系列修改，不再是所有的字符串都会被符号化了，因此我们还需要修改字符串比较的部分。之前只需要比较字符串的地址就能判断字符串是否相同，而今后就需要比较没有符号化的字符串的内容了（图 2-53）。这种做法的关键之处在于为符号化的字符串设置 `STRM_STR_INTERNED` 标志位。

```
int
strm_str_eq(strm_string *a, strm_string *b)
{
  /* 如果地址相同则字符串相同 */
  if (a == b) return TRUE;
  /* 如果两个字符串都已符号化 */
  if (a->flags & b->flags & STRM_STR_INTERNED) {
    /* 地址不同就意味着对象不同 */
    return FALSE;
  }
  /* 从这里开始是对内容的比较 */
  /* 如果长度不同则字符串不同 */
  if (a->len != b->len) return FALSE;
  /* 比较内容，如果内容相同则字符串相同 */
  if (memcmp(a->ptr, b->ptr, a->len) == 0) return TRUE;
  /* 不相同 */
  return FALSE;
}
```

图 2-53　字符串比较函数

语法分析动作

接下来我们回到语言处理的实现上。

之前介绍过如何使用 yacc 工具定义语法。把 yacc 的语法定义文件传给 yacc 工具，yacc 工具就会帮我们生成语法分析的函数。

在语法定义中增加“动作”，就可以根据语法来执行相应的处理。动作是指规则匹配时执行的代码。

后面我们会看到，在动作部分，“`$$`”是该规则生成的值，“`$1`”等是语法的第 n 个元素生成的值。需要注意的是，语法规则匹配时动作会被立即执行，因此执行顺序可能与预想的不同。我们把它理解为事件驱动可能更为合适。

转换为抽象语法树

也就是说，语言处理的本质是在动作部分写处理代码。如果想让语言处理达到一定的复杂程度，就需要编写相应的动作代码。

许多语言处理器会在动作部分将程序转换为树结构。这是因为转换前的程序只是一些文本，处理起来比较困难。

把程序转换为树结构之后的结构被称为抽象语法树（Abstract Syntax Tree，AST），比如 Streem 中相当于 Hello World 的“将从标准输入读取到的字符串写到标准输出”这一简单程序的抽象语法树就如图 2-54 所示。

带有运算符的表达式（op）用 node_op 节点表示，它的下面有 3 个分支。一个是保存运算符名的 op 分支，这里运算符名是“|”。参数保存在第 2 个分支（lhs）和第 3 个分支（rhs）中。

运算符之外的表达式和语句也有相应的节点，比如函数调用的节点是 node_call，if 语句的节点是 node_if。

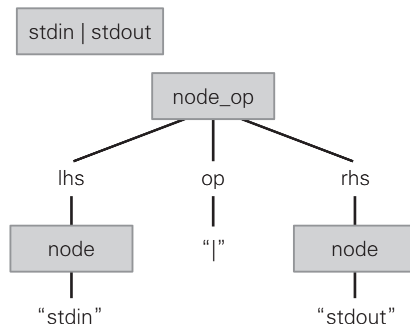


图 2-54 抽象语法树的例子

lhs 和 rhs 分别是“left hand side”（左边）和“right hand side”（右边）的缩写

用结构体表示语法树的节点

表示抽象语法树的结构体的定义如图 2-55 所示。

```
typedef enum {
  NODE_ARGS,
  NODE_PAIR,
  NODE_VALUE,
  NODE_CFUNC,
  NODE_BLOCK,
  NODE_IDENT,
  NODE_LET,
  NODE_IF,
  NODE_EMIT,
  NODE_RETURN,
  NODE_BREAK,
  NODE_VAR,
  NODE_CONST,
  NODE_OP,
  NODE_CALL,
  NODE_ARRAY,
  NODE_MAP,
} node_type;

#define NODE_HEADER node_type type

typedef struct {
  NODE_HEADER;
  node_value value;
} node;

typedef struct {
  NODE_HEADER;
  node* recv;
  node* ident;
  node* args;
  node* blk;
} node_call;

// 下面是其他结构体的定义
```

图 2-55 抽象语法树的结构体

组成抽象语法树节点的结构体，其前面有一个名为 node_type type（NODE_HEADER）的共同成员。程序通过指针访问 node 结构体，程序员根据需要参考 type 的值进行类型转换。从类型安全的角度来看，这段代码的写法很糟糕，但这是动态类型语言中常用的一个技巧。

接下来定义创建与语法相应的节点的函数（图 2-56），然后在动作部分进行调用即可（图 2-57）。

```
extern node* node_array_new();
extern node* node_pair_new(node*, node*);
extern node* node_map_new();
extern node* node_let_new(node*, node*);
extern node* node_op_new(const char*, node*, node*);
extern node* node_block_new(node*, node*);
extern node* node_call_new(node*, node*, node*, node*);
extern node* node_int_new(long);
extern node* node_double_new(double);
extern node* node_string_new(const char*, size_t);
extern node* node_if_new(node*, node*, node*);
extern node* node_emit_new(node*);
extern node* node_return_new(node*);
extern node* node_break_new();
extern node* node_ident_new(node_id);
extern node* node_ident_str(node_id);
extern node* node_nil();
extern node* node_true();
extern node* node_false();

// 相当于if语句的节点
node*
node_if_new(node* cond, node* then, node* opt_else)
{
  node_if* nif = malloc(sizeof(node_if));
  nif->type = NODE_IF;
  nif->cond = cond;
  nif->then = then;
  nif->opt_else = opt_else;
  return (node*)nif;
}

// 相当于整数的节点
node*
node_int_new(long i)
{
  node* np = malloc(sizeof(node));
```

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

```
  np->type = NODE_VALUE;
  np->value.t = NODE_VALUE_INT;
  np->value.v.i = i;
  return np;
}

// 下面是其他同样生成节点的函数的定义
```

图 2-56 节点生成函数（摘录）

```
program    : compstmt
               { /* 将生成的节点 */
                 /* 保存在parser_state p中 */
                 p->lval = $1;
               }
           ;

/* 中间省略 */

primary0   : lit_number /* 节点在lex.l中生成 */
           | lit_string /* 同上 */
           | identifier
               {
                 $$ = node_ident_new($1);
               }
           | '(' expr ')'
               {
                 $$ = $2;
               }
           | '[' args ']'     /* 链表 */
               {
                 $$ = node_array_of($2);
               }
           | '[' ']'          /* 空链表map */
               {
                 $$ = node_array_of(NULL);
               }
           | '[' map_args ']' /* map */
               {
                 $$ = node_map_of($2);
```

```
            }
        | '[' ':' ']'       /* 空map */
            {
              $$ = node_map_of(NULL);
            }
        | keyword_if condition '{' compstmt '}' opt_else
            {
              $$ = node_if_new($2, $4, $6);
            }
        | keyword_nil
            {
              $$ = node_nil();
            }
        | keyword_true
            {
              $$ = node_true();
            }
        | keyword_false
            {
              $$ = node_false();
            }
        ;
```

图 2-57 创建抽象语法树的动作（摘录）

实际的源代码在 Streem 源码仓库（https://github.com/matz/streem）的 src 目录下，大家可以参考该目录下的 parse.y（语法分析部分的 yacc 源代码）和 node.c（节点生成部分）。

这样一来，执行用 yacc 生成的 `yyparse` 函数，就能得到抽象语法树了。

直接执行语法树

语法分析结果的抽象语法树生成之后，就该对其进行处理了。处理步骤有很多，各个语言处理器的处理方式也不尽相同（表 2-3）。

表 2-3 各语言处理器对抽象语法树的处理

语言处理器	处理
Ruby 1.8	直接执行抽象语法树
Ruby 1.9 以后	生成虚拟机机器码之后在虚拟机上执行
mruby	生成虚拟机机器码。也可以直接在虚拟机上执行
Python	生成虚拟机机器码之后在虚拟机上执行
Java	生成虚拟机机器码

从表 2-3 来看，从抽象语法树生成虚拟机机器码（习惯上称之为字节码）的语言处理器占大多数。

这么做有它的道理。因为从内存访问的效率等角度来看，比起遍历抽象语法树的链接来执行，

一般情况下一次性生成字节码后在虚拟机上执行的效率会更高。Ruby 在 1.9 版本之后性能得到了大幅提升，原因就在于引入了虚拟机。

那么 Streem 应采用哪种处理方式呢？当然，最终我会引入某种形式的虚拟机，不过实现虚拟机也需要花费相应的时间和精力。为了尽早实现让 Streem 运行起来的目标，我决定暂且与 Ruby 1.8 一样编写直接遍历抽象语法树的执行函数。

也许有人觉得这是在浪费时间，但很多时候如果不实际运行起来看看，就无法想象所编写的语言具体是什么样子的。即使是为了反复推敲语言的细节，也需要尽早让语言进入可运行的状态，这是非常重要的。而且没有什么比尝试让自己写的代码实际运行起来更能让程序员兴奋的了，这是程序员动力的源泉。现在回想起来，在 Ruby 二十多年的开发过程中，最痛苦的时期就是从开始开发到 Ruby 实际能跑起来为止的那半年。

遍历抽象语法树

语法分析器把程序的文本转换为表示抽象语法树的结构体的链表结构。在解释程序时，需要遍历这个链表结构。

一边遍历 Streem 的抽象语法树一边运行的函数是 exec.c 文件中的 `exec_expr()` 函数。这个函数用一个很长的 `switch` 语句来根据节点的种类进行相应的处理。`exec_expr` 函数非常长（133 行），所以这里只摘录其中一部分（图 2-58）。

```
/* 执行抽象语法树的函数 */
/* ctx: 上下文 */
/* np:  抽象语法树 */
/* val: 执行结果 */
/* 返回值: 0 – 成功, 1 – 失败 */
static int
exec_expr(node_ctx* ctx, node* np, strm_value* val)
{
  int n;

  /* 抽象语法树如果为NULL则失败 */
  if (np == NULL) {
    return 1;
  }

  /* 根据抽象语法树的类型开始分支处理 */
  switch (np->type) {
  /* 访问变量 */
  case NODE_IDENT:
```

```
    /* 根据变量名取出值 */
    *val = strm_var_get(np->value.v.id);
    return 0;
  /* if语句 */
  case NODE_IF:
    {
      strm_value v;
      /* 把np类型转换为node_if */
      node_if* nif = (node_if*)np;
      /* 执行条件部分（递归调用exec_expr） */
      n = exec_expr(ctx, nif->cond, &v);
      /* 条件部分执行失败则处理失败 */
      if (n) return n;
      /* 如果条件部分为真 */
      if (strm_value_bool(v)) {
        /* 执行then部分 */
        return exec_expr(ctx, nif->then, val);
      }
      else if (nif->opt_else != NULL) {
        /* 执行else部分 */
        return exec_expr(ctx, nif->opt_else, val);
      }
      else {
        /* 如果没有else部分则为null */
        *val = strm_nil_value();
        return 0;
      }
    }
    break;
  /* 运算符表达式 */
  case NODE_OP:
    {
      /* 类型转换为node_op */
      node_op* nop = (node_op*)np;
      strm_value args[2];
      int i=0;

      /* 执行左边 */
      if (nop->lhs) {
        n = exec_expr(ctx, nop->lhs, &args[i++]);
        if (n) return n;
```

```
        }
        /* 执行右边 */
        if (nop->rhs) {
          n = exec_expr(ctx, nop->rhs, &args[i++]);
          if (n) return n;
        }
        /* 函数调用（调用名称为“|”的函数） */
        return exec_call(ctx, nop->op, i, args, val);
      }
      break;
    /* 函数调用 */
    case NODE_CALL:
      ...
    }
  }
```

图 2-58 exec_expr 函数（摘录）

灵活应用递归调用

在实现遍历这种树结构的函数时，一般使用递归调用，比如运算符表达式会按照以下顺序执行。

1. 递归调用左边的部分去执行
2. 递归调用右边的部分去执行
3. 把两边的运行结果作为参数，调用相当于运算符的函数

其他表达式和语句也是一样的。由于需要为每种类型的节点编写处理代码，所以代码会变得很长，不过做的只是重复相同的处理，其实也没那么复杂。

同样通过递归调用树结构来进行遍历的还有 main.c 的 `dump_node()` 函数。这个函数是在调试时使用的，用缩进表示树结构，代码的结构与 `exec_expr()` 相似。图 2-59 是 `dump_node()` 函数的摘录。将图 2-54 的树结构用 `dump_node()` 函数输出，代码就如图 2-60 所示。

```
/* 打印抽象语法树的函数 */
/* np: 抽象语法树 */
/* indent: 缩进层级 */
static void
dump_node(node* np, int indent) {
  int i;
```

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

```
/* 缩进到指定的层级为止 */
for (i = 0; i < indent; i++)
  putchar(' ');

/* 如果为NULL则打印NIL */
if (!np) {
  printf("NIL\n");
  return;
}

/* 根据抽象语法树的类型开始分支处理 */
switch (np->type) {
/* if语句 */
case NODE_IF:
  {
    /* 打印类型 */
    printf("IF:\n");
    /* 打印条件部分 */
    dump_node(((node_if*)np)->cond, indent+1);
    for (i = 0; i < indent; i++)
      putchar(' ');
    printf("THEN:\n");
    /* 打印then部分 */
    dump_node(((node_if*)np)->then, indent+1);
    node* opt_else = ((node_if*)np)->opt_else;
    /* （如果不为空）打印else部分 */
    if (opt_else != NULL) {
      for (i = 0; i < indent; i++)
        putchar(' ');
      printf("ELSE:\n");
      dump_node(opt_else, indent+1);
    }
  }
  break;
/* 运算符表达式 */
case NODE_OP:
  /* 打印类型 */
  printf("OP:\n");
  for (i = 0; i < indent+1; i++)
    putchar(' ');
  /* 打印运算符名 */
```

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

```
    print_id("op: ", ((node_op*) np)->op);
    /* 打印左边 */
    dump_node(((node_op*) np)->lhs, indent+1);
    /* 打印右边 */
    dump_node(((node_op*) np)->rhs, indent+1);
    break;
    /* 中间省略 */
    ....
  default:
    /* 未知类型（错误） */
    printf("UNKNWON(%d)\n", np->type);
    break;
  }
}
```

图 2-59　dump_node() 函数（摘录）

```
stdin | stdout

STMTS:
 OP:
  op: |
  IDENT: stdin
  IDENT: stdout
```

图 2-60　抽象语法树的 dump 输出

用开源的方式开发

本节介绍的抽象语法树的生成和运行部分是在 mattn 先生提交的 Pull Request 的基础上开发的。当我还在开发之前讲解的事件循环的时候，他就已经帮我把基础部分的代码写好了。这就是开源的力量。

但这并不是说发给我的代码直接就能用。包括修改函数和结构体的名称在内，我做了大量修改。拥有普通软件开发（开源以外的）经验的人也许会觉得这有些奇怪，因为在多数情况下“应该避免”多人修改同一处代码的情况，而且有些人也会因此而感到不舒服。

对自己编写的代码有感情，不希望别人修改是人之常情，但许多开源项目并不怎么看重这种感情。

在开源项目中，很多人提交一次修改之后就不再出现了，如果过于看重“所有权意识”和“负责人意识”，那么开发就不会进步。第三者自由地发送修改申请，作者采纳修改意见，这是开源软件中典型的一种行为，如果原作者的所有权意识太强，那是无法接受这种行为的。在开源项目中，或许可以说所有权意识这种“自我意识”不被看重才是自然的。

理想的语言处理器

对照 Ruby 1.8 就会发现，本节讲解的遍历抽象语法树的处理器虽然实现起来很简单，但是性能不好，最大的原因在于 2-6 节讲解的内存缓存。

遍历结构体的链接就意味着要依次访问分散在内存中的结构体，所以缓存的效率可以说是最差的。

为了有效利用内存缓存，一次访问尽量在有限的内存空间内连续进行是比较理想的，因此许多语言处理器选择将抽象语法树转换为虚拟机的指令序列，然后解释这个指令序列去实际运行。

我打算让 Streem 也采用这种方式。因为 Streem 的使用场景决定了性能是不可忽视的一个要素。我还想进一步尝试实现 JIT（Just-in-time）编译器，在运行时生成机器码并运行。

但是在现阶段，把精力放在语言设计上更为重要。有一句程序员格言是这么说的：“过早优化是万恶之源。”

今后的计划

现在终于有运行 Streem 程序的感觉了，但是 Streem 还不能定义函数，也不能进行管道操作，所以只是我心里这么感觉而已。

因此下一步要实现函数运行的部分，让“完整”的 Streem 程序能够运行起来。另外，我也准备和大家探讨一下异常处理的相关内容，从而应对运行中发生的错误。

小结

通过组合前面在 parse.y、node.c 和 exec.c 中定义的函数，我们可以轻松地创建姑且能够运行起来的语言。本节对应的源代码的标签是 `201506`。大家可以参考这个源代码去试着创建自己的语言。

对语言设计进行各种思考之后，就可以想象出自己正在使用的语言为什么是现在这个样子的、语言设计者是怎么想的，等等，非常有意思。

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

时光机专栏

语法树的实现方法不止一种

本节是 2015 年 6 月刊中刊登的内容，主要讲解了连接语法分析器（前端）和代码生成部分（后端）的语法树的实现。

语法树的实现方法，或者说连接前端和后端的方法不止一种（而且也不限于树结构），其中还有不区分前后端，在语法分析器中直接生成代码的编译器。不过在语法分析器中直接生成代码的做法不利于实施优化，因此我并不推荐这么做。

很多编译器会将程序语法解析为某种数据结构，然后传给代码生成部分。而树结构在表示程序的数据结构中是比较常用的，所以反映了语法的树结构，也就是语法树经常被各种语言使用。

语法树的实现方式多种多样。我参与的语言处理器中，本次介绍的 Streem 根据节点种类的不同使用了不同的结构体，mruby 则通过与 Lisp 类似的 `cons cell` 的链接来创建树结构。另外，CRuby 在表示节点的结构体中使用 `union` 来区分节点的种类。如上所示，我们不能说哪种方式才是正确的。

顺便说一句，在传统的编译器 gcc 和 clang 中，gcc 使用了 RTL（Register Transfer Language）、clang 使用了 LLVM IR（Intermediate Representation）这种中间代码的形式来连接前端和后端。这两种方式都不是语法树数据，真是有趣。

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

2-8 局部变量与异常处理

Streem可以作为语言开始运行了，这次我们再为它添加两个功能：一个是局部变量，在这部分内容中我们将探讨是否允许嵌套、如何实现闭包等设计上的问题；另一个是异常处理，关于异常处理，我们将探讨如何通过忽视错误以让处理继续进行。

经过 2-7 节的处理，Streem 终于有编程语言的样子了，这次我们再为它增加局部变量和异常处理两个功能。

■ 局部变量

我们首先来思考一下局部变量的相关内容。对现在的编程语言来说，局部变量可以说是常识中的常识，但在很久之前却并非如此。

回到 30 年前

我们回到 30 年前看一看。那时编程爱好者们常用的语言是 BASIC，而当时的 BASIC 语言中没有局部变量。大家能想象出没有局部变量的编程场景吗？所有变量在任何地方都有可能被修改，因此很难知道值是在哪里被修改的。

当时，年轻的开发者们都是用 BASIC 语言来开发程序的，其中就不乏游戏等具有一定规模的程序。这样的程序竟然都能调试，现在想想真是让人敬佩。

没有局部变量的世界

BASIC 中是用下面这样的行号来调用子程序的。

```
gosub 4000
```

由于没有参数和返回值，所以只能用全局变量来传递值。具体做法是把值保存在某个变量中，调用子程序，然后计算结果就会被保存到另一个变量中。当然，像信息隐藏这种高级功能是无法实现的，把一个流程汇总到一个函数中进行抽象化也是不行的。

甚至连函数都没有，所以也无法进行函数的递归调用。

局部变量的引入

要说最早发明了局部变量的编程语言是什么，很遗憾我没有查到准确的信息，但可以确认最早引入局部变量的语言是 Algol。Algol 虽然没有像 FORTRAN 语言那样流传到现在，但它引入的语言特性却影响了后来的很多语言。

有了局部变量就可以把一系列处理封装起来作为函数提供，这也使得递归调用成为可能（图 2-61）。不过，如果强行使用数组自己去开发栈，那么用全局变量进行递归调用也不是不可能。据说 FORTRAN 在还没有局部变量的时候就是使用这种方式编程的，不过我不想这么做。

```
def fact(n)
  if n == 1
    1
  else
    n * fact(n-1)
  end
end
```

图 2-61　递归调用

局部变量的实现

局部变量的实现并不是很难，比如可以在函数编译时为每个局部变量分配一个不同的索引，在函数运行时准备一个数组，将值保存在该变量的索引指定的位置。函数在每次运行时使用的数组一般称为栈。

这里有一点需要注意，就是要把握好函数每次运行时使用的局部变量的数量，不要超过栈的大小。栈溢出在安全层面上也是一个重大的问题。

接下来我们就尝试在 Streem 语言处理器中增加局部变量功能。首先在语法分析部分修改变量的赋值和访问的代码。

在 Streem 中实现局部变量

现在的 Streem 语法分析器在赋值部分生成 `NODE_LET` 节点，在访问部分生成 `NODE_IDENT` 节点。

之前的实现中都没有涉及 `NODE_LET`，在访问全局变量时使用了 `NODE_IDENT`。我们来修改一下这部分内容。首先在 `NODE_LET` 初始化局部变量。刚才介绍过要为每个局部变量分配索引，由于这个版本的解释器不考虑性能方面的问题，所以局部变量也用散列表保存。将来引入虚拟机之后我们再考虑性能问题。

对于 `NODE_LET`，需要进行散列表的初始化和赋值（如果还没做的话）。在 Streem 中，即使是局部变量也只能赋值一次，不能再做修改，因此当已赋值的局部变量再次被赋值时，就会抛出运行时错误。当然，将来引入虚拟机时就应该是编译错误。

`NODE_IDENT` 访问局部变量表，如果其中存在要访问的变量，就取出变量的值，否则就访问全局变量表，如果没有定义全局变量则报错。

Streem 在函数运行时会传给函数 node_ctx 结构体。这个结构体保存运行时的上下文（语境），用于实现局部变量的局部变量表（散列）要作为成员变量添加到这个结构体中。

node_ctx 只是用于遍历现在的节点并执行的结构体，所以将来引入虚拟机时应该会修改结构体的名称。

局部变量的嵌套

语言设计上需要决定是否允许局部变量嵌套，对此，各个语言的规则不尽相同。比如 C 和 Java 允许局部变量嵌套，在括号括起来的范围内定义的局部变量只在该作用域的范围内有效。在不同的作用域，即使定义同名的变量，也会被当成不同的变量（图 2-62）。

可以在内侧的作用域定义与外部作用域的变量同名的变量。因为是同名的不同变量，所以即使名称相同，类型也可以不同。不过从代码可读性的角度考虑，还是不要这么做比较好，以免造成混乱。

而 Ruby 只是在类定义和方法定义中引入了作用域，没有准备像 C 的括号那样的临时作用域的语法（除了后面要介绍的特殊情况）。

```
void
func()
{
  int i = 10; /* i的作用域是整个func函数 */

  while (i--) {
    int j = 5; /* j的作用域限定在while中 */

    printf("i:%d j:%d\n", i, j);
  }

  /* 用括号引入新的作用域 */
  {
    double j = 1.5; /* 这个变量与上面的j不同 */

    printf("new j:%g\n", j);
  }
}
```

图 2-62　C 的嵌套作用域

前面也说过，通过嵌套作用域定义同名的不同变量并不是一种不可缺少的语言特性（只需定义一个其他名字的变量即可避开这一特性），它反而会带来混乱，所以 Ruby 没有特意去引入。

关于这一点，我准备在 Streem 中也采用相同的做法。

作用域嵌套的特殊情况

前面说到 Ruby 除了特殊情况以外没有引入嵌套的作用域，我们接下来就来看一看特殊情况到底指什么。

Ruby 在设计上确实避免了作用域嵌套，但闭包（closure）却是个例外。一般来说，在类或方法内定义的变量的作用域范围就是在该类或方法的范围内，而在类或方法内的代码块或匿名函数中

出现的变量的作用域范围则只是在该代码块或匿名函数的范围内（图 2-63）。

思考一下就可以明白，由于 Ruby 的代码块和匿名函数是函数，所以需要使用限制了访问作用域的局部变量，不然就要倒退回只有全局变量的子程序时代了。

```
# 从do开始到end结束的代码块是作用域范围
[1,2,3].each do |i| # i只在代码块内有效
  sq = i * i        # sq也只在代码块内有效
  p [i, sq]
end

# 匿名函数也引入了作用域
f = ->(x) { x * x } # x只在代码块内有效
```

图 2-63 Ruby 的嵌套作用域

p 是将参数的对象转换成易读的字符串，然后打印到标准输出的方法

为了减少混乱，我做了一点改进。在使用像 C 和 Java 那样的嵌套作用域时，对与外部作用域同名的变量发出警告，但这样做的缺点是局部变量的有效范围不再一目了然。就现状来说，这并不是最理想的改进方法，而且还与 Ruby 没有变量声明的语法相抵触（图 2-64）。

```
# 碰巧与外部作用域变量的变量名重复了
e = 10
[1,2,3].each do |i|
  p i
end

[1,2,3].each do |e|
  p e
end
# 指定-v参数时
# 会被警告
#  warning: shadowing outer local variable - e

# 从作用域中拿出变量比较麻烦

even = nil # 如果忘记了这行就会出错
[1,2,3].each do |i|
  if i % 2 == 0
    even = i
  end
end
p even     # 如果没有事先进行初始化则不能访问even
# 指定-v参数时
# 会被警告
#  warning: assigned but unused variable - even
```

图 2-64 Ruby 局部变量的缺点

我个人认为这是 Ruby 的设计中令人不太满意的地方。过去为了改进这个问题我想了很多办法，图 2-64 中的警告等就是一个反映。实际上我想实现的不仅仅是警告，还想通过设计语言的作用域来进一步做出改善，但考虑到兼容性的问题，也担心语言会变得过于复杂，所以最终没有实现。

没有采用的语法规则还包括局部变量的传递等（图 2-65）。

```
[1,2,3].each do |i|
  if i % 2 == 0
    # 这里被初始化的变量
    even = i
  end
end
# 如果在作用域外被访问，则提升它的作用域
# 也就是说，把它当作属于外部作用域的局部变量
p even
```

图 2-65　局部变量的传递

这虽然是一个好方法，但实现起来比较麻烦，而且还可能会带来更严重的问题

闭包（函数闭包）

代码块中允许局部变量的嵌套就意味着可以从代码块和匿名函数访问外部作用域的变量，而且有时匿名函数出了作用域之后还能继续存活。

比如图 2-66 的程序。在函数 `incdec` 内部创建的两个匿名函数分别访问了外部的局部变量 `acc`。`incdec` 运行结束后，一般来说这时局部变量应该被回收了，但由于它还在被匿名函数使用，所以就没有被回收。外部作用域的变量被“封闭”在函数对象之中，所以我们就把这种状态称为闭包或者函数闭包。

```
def incdec
  acc = 0 # 被封在里面的变量
  inc = ->() {
    acc += 1
    acc
  }
  dec = ->() {
    acc -= 1
    acc
  }
  return [inc, dec]
end

inc, dec = incdec()
p inc.call  # => 1  acc加1
p inc.call  # => 2  acc加1
p dec.call  # => 1  acc减1
p dec.call  # => 0  acc减1
```

图 2-66　闭包

闭包的实现

这种闭包实现起来非常麻烦。Ruby 的处理器把局部变量的嵌套关系作为“环境”保存在函数对象中，这样在访问作用域外部的函数时，使用的就是外部的环境。

mruby 中的函数对象（proc）与环境（env）的定义如图 2-67 所示。

mruby 虚拟机访问外部作用域的指令有 OP_GETUPVAR 和 OP_SETUPVAR，两者都可以得到操作数来指定访问第几层外部作用域的第几个变量。

环境是单独的 Ruby 对象，在被函数对象使用期间保持存活。如果没有对象使用，则由垃圾回收器回收。

```
struct REnv {
  MRB_OBJECT_HEADER;
  mrb_value *stack;
  mrb_sym mid;
  ptrdiff_t cioff;
};

struct RProc {
  MRB_OBJECT_HEADER;
  union {
    mrb_irep *irep;
    mrb_func_t func;
  } body;
  struct RClass *target_class;
  struct REnv *env;
};
```

图 2-67　mruby 的闭包实现

Streem 的闭包

但是 Streem 与 Ruby 有一处不同，这就使得 Streem 在闭包的实现上更加简单。这个不同之处就是，Streem 中即使是局部变量也不允许被修改。当然，这一点也导致了 Streem 不能创建图 2-66 那样的有状态的闭包，但考虑到状态和副作用是函数式语言应该避开的，所以也就算不上什么不好的事情了。

Streem 的函数对象的定义如图 2-68 所示。如前所述，在 Streem 中不需要担心局部变量被修改，所以闭包直接复制变量的值即可。不过现在的实现还很简单，内部还保持着外部环境的链接，每次访问时都会遍历链接。而虚拟机版本为了改善性能，在生成函数对象时会复制变量的值。

```
typedef struct strm_lambda {
  STRM_OBJ_HEADER;
  /* 函数的主体 */
  struct node_lambda* body;
  /* 保持作用域的上下文 */
  struct node_ctx* ctx;
} strm_lambda;
```

图 2-68　Streem 的闭包实现

编译时检查

这次我们实现了对已经存在的局部变量再次赋值，以及访问不存在的局部变量时抛出运行时错误的功能，但其实从程序字面上就可以判断出对局部变量的赋值和访问是否报错，所以本来应该算作编译错误的。

运行时错误意味着“不运行就不会被发现”，让人无法放心，但编译错误就没有这方面的问题。在编程语言的发展历程中，以前的语言就连语法错误都在运行的时候去检查，但现在更多的是在编译时检查，所以我打算近期改善这个问题。

■ 异常情况

介绍完局部变量，下面我们来思考一下异常处理。

在进行各种处理时，某些原因可能会使处理没有按照预期进行。尽管自己很想把精力集中在处理上，但也无法对这些异常情况视而不见。

我们就以“打开文件”这个简单的处理为例来思考一下。需要做的仅仅是“指定文件名，然后打开文件”，在 Linux 等类 UNIX 操作系统中会使用 open 系统调用完成这个处理。

open 系统调用的原型如图 2-69 所示。flags 参数用于指定文件打开模式（读取、写入和追加其中之一），当创建新文件时，使用第 3 个参数指定文件模式，也就是文件的访问权限。

```
int open(const char* path, int flags);
int open(const char* path, int flags, mode_t mode);
```

图 2-69　open 系统调用

即便是如此简单的文件打开处理，也会发生异常情况。表 2-4 汇总了 open 系统调用可能会发生的错误。包括 EPERM 这种 Linux 特有的错误在内，实际上可能发生的异常情况竟有 23 种。

当然在大多数情况下文件能够正常打开，但这并不意味着就可以忽视异常情况。异常情况出现时，程序会异常终止，这种情况反而会给我们带来困扰。在使用文本编辑器编辑文件时，如果输错了文件名，选择了不存在的文件，使整个文本编辑器异常退出，那就让人欲哭无泪了。

也就是说，软件运行时肯定会伴随着异常情况，所以需要对其进行恰当的处理。

但另一方面，异常情况终究只是个例，我们一般不太想去编写处理代码，也不想去读这些代码。插入了异常处理的代码之后，导致程序本身的逻辑变得让人费解，这是我们不希望看到的。

对语言设计来说，如何处理异常情况是非常重要的。

表 2-4　open 系统调用可能发生的错误

错误	内容
EACCES	没有文件访问权限
EDQUOT	已到达 disk quota（容量限制）
EEXIST	文件已存在
EFAULT	path 是不正确的地址
EINTR	系统调用被中断
EINVAL	指定了不正确的 flags
EISDIR	指定的是目录
ELOOP	符号链接循环
EMFILE	进程打开了过多的文件
ENAMETOOLONG	文件名过长
ENFILE	系统打开了过多的文件
ENODEV	设备不存在
ENOENT	文件不存在
ENOMEM	（内核空间的）内存不足
ENOSPC	磁盘已满
ENOTDIR	path 不是目录
ENXIO	没有 FIFO 管道操作的对象
EOPNOTSUPP	不支持 tmpfile
EOVERFLOW	文件过大
EPERM	没有 O_NOATIME 权限
EROFS	尝试在只读磁盘写入
ETXTBSY	没有向运行中的程序写入的权限
EWOULDBLOCK	指定了 O_NONBLOCK 时可能会阻塞

错误检查

C 语言以及最近的 Go 语言都会进行错误检查，比如在调用有可能会失败的函数时，这些语言会检查返回值，确认函数是否执行成功。

这种做法的优点是不需要在语法层面上做任何支持，所以实现起来非常简单。下面要介绍的“异常”是在你没注意到的时候程序中止了运行，因此根据中止时间点的不同可能会有意料之外的情况发生。比如可能会出现数据不一致的问题，或者没有释放内存，发生内存泄漏等。

（有异常机制的）C++ 把不会发生这种问题的情况称为“异常安全”，但是稍微了解一下就能知道，在 C++ 中保证“异常安全”也是非常困难的。而错误检查就避免了这样的困难。

但是错误检查有一个缺点，就是异常情况的处理代码与主逻辑的代码混在一起，正常情况的处理代码被埋没，从而使程序变得令人费解。如果忘记进行错误检查，程序在前提条件不成立的情况下仍强制运行，最终就可能导致异常退出，甚至引起安全方面的问题。

Go 语言利用函数可以返回多个值这一特点来降低忘记检查的风险，但代码依然很烦琐。

让“异常”产生

接下来的说法可能容易和前面的内容混淆：在发生异常情况时，很多编程语言提供了产生“异常”对象、中止程序运行的功能。C++、Java 和 Ruby 等都提供了这种异常机制。在现在的编程语言中，相比进行错误检查，提供异常机制的语言更加普遍。

异常最大的优点是，当异常情况导致前提条件不成立时（比如文件不存在导致无法打开等），程序会自动中止运行，因此程序就不会在前提条件不成立的情况下继续运行，从而也就保障了安全。

异常的缺点我们在前面也提到过，由于发生异常情况时程序会自动中止运行，所以很难保证异常安全。但是支持垃圾回收的 Ruby 等语言要比手动管理资源的 C++ 等语言更容易维持异常安全，所以说实现的困难程度因语言而不同。

Swift 的 Optional

美国苹果公司开发的 Swift 没有提供异常处理机制，取而代之的是，Swift 将可能会失败的函数的类型指定为 `Optional<T>` 类型。`Optional` 这种类型既可以存储某种类型的任意一个值，也可以为 `nil`（空），大部分函数在失败时会返回 `nil`。Swift 中可以把 `Optional<T>` 简写为 `T?`。

当某种类型被 `Optional` 包裹时，不能直接使用这个类型的值。

```
var i: Int? = 10;
println(i + 2) // 错误
```

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

Swift 中把从 Optional 类型中取出实际的值的操作称为 unwrap。在变量名后加上“!”就可以取出值，但值为 nil 时会发生运行时错误。

```
println(i! + 2)
```

可以在值为 nil 时指定要取的值。

```
println((i??5) + 2)
```

或者通过组合使用 let 和 if 显式检查 nil。

```
if let i2 = i {
  println(i2 + 2)
}
```

这段代码里的 i2 的类型不是 Optional，而是 Int，所以不需要再去 unwrap。

最后介绍一下使用“?.”代替“.”来调用方法的语法。使用“?.”时，如果值不为空，就执行这个值的方法，如果为 nil，则什么也不执行，直接返回 nil。

```
var dog? = Dog()
dog!.bark() // 为nil时错误
dog?.bark() // 为nil时返回nil
```

像这样，Swift 不通过异常机制，而用 Optional 类型来处理异常情况（错误）。我认为这个 Optional 类型参考了 Haskell 的 Maybe 类型和 OCaml 的 Option 类型，是一个巧妙利用静态类型进行错误处理的好办法。

忽视错误

下面来思考一下 Streem 中如何进行异常处理。与其他语言不同，Streem 有两个明确的运行阶段，即准备管道的初始化阶段和数据在管道中流转的管道阶段。在初始化阶段如果发生异常情况，之后的处理将会很难继续运行，所以我决定使用普通的异常机制。

而管道阶段中会有大量数据流转。我们不希望在读取 10 GB 的数据文件时，因其中 1 行数据的损坏而导致整个处理失败。

所以在管道阶段除非显式地进行了指定，否则只会中止运行发生错误的数据，管道继续进行后续的数据处理。美国谷歌公司开发的云上的数据处理语言 Sawzall 也采用了这种忽视错误继续处理的做法。

Streem 的异常处理的实现

实现 Streem 方法的 C 函数的原型如图 2-70 所示。argc 和 argv 是参数，ret 表示方法的返回值。方法的返回值代表方法运行的结果：成功时为 0，失败时返回 0 以外的值。

```
int exec_plus(node_ctx* ctx, int argc, strm_value* args, strm_value* ret);
```

图 2-70　实现 Streem 方法的函数

小结

不管是局部变量还是异常处理，都是现代编程语言必备的功能，但从语言设计的角度来看，这么寻常的功能也有很多需要解决的问题和需要权衡的地方。

语言的设计就是在这种细微之处不断推敲的一种行为。

时光机专栏

看似很长其实很短的编程语言的历史

本节是 2015 年 7 月刊中刊登的内容，介绍了相对独立的局部变量和异常处理这两部分内容。

在现在的编程中，局部变量是常识一样的存在，令人惊讶的是，三十几年前没有局部变量的编程语言到现在还在被人使用。就连实际经历过那个年代的我，在重温那段历史时还是会感到吃惊，所以一开始学的就是“语法完备”的编程语言的人可能会更加震惊。

在历史尚短的编程领域，这样的事情时常发生，以为是很久很久以前的事情其实只发生在几十年前，以为是历史上的人物其实还健在。

关于异常处理我也再说几句。异常机制应该是在 Lisp 或者与其相关的语言中诞生的，所以有四十多年的历史，但在 Java 普及之前一直都没有被广泛使用，这一点与垃圾回收相似。

有了异常机制之后，错误处理的描述变少了，程序的可读性更强了，这是它的优点。但异常机制也不全是优点，程序有可能在意料之外的时间点中止处理，所以可能会有预料之外的情况出现。

Go 以异常处理不适合在并发编程中使用为由，没有引入异常机制，而是采用了错误检查的方法。Streem 的异常处理原则是发生错误时丢掉那部分数据，灵活利用了流处理的特性。如果是预料之外的数据引起的错误倒也没什么问题，可如果忽略了程序的 bug 所引起的错误，就可能在调试上出现困难，所以今后还需要去改进这一点。

第 3 章

设计面向对象功能

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

3-1 各种各样的面向对象

面向对象编程是什么？对此大家众说纷纭，现在还没有一个明确的定义。这次我们就根据历史来聊聊面向对象的话题（以干货为主）。

在 1993 年即将开始开发 Ruby 的时候，我和同事石塚圭树（之后的 Ruby 的命名者）一起策划写一本书。石塚已经通过 ASCII 出版社出版了一本名为《面向对象编程》[①] 的书，他想再写一本书作为这本书的续集。

计划写的内容用一句话概括就是“通过创建面向对象语言来学习面向对象编程”，目的是让读者在设计和实现语言处理器的过程中学习面向对象编程的思想，从而深入地理解面向对象编程。

遗憾的是这个策划因为“看起来很难卖出去”而流产了，而当时打算作为图书示例编写的语言则成为了后来的 Ruby，想必没什么人知道还有这段故事。

面向对象语言原本就是按照设计者的意图和想法开发的，所以通过语言来学习面向对象编程的想法也没有那么糟糕。

尽管篇幅有限，但这次是个很难得的机会，我打算在二十多年后的今天再次挑战通过语言学习面向对象这一主题。首先从最早的面向对象语言开始。

Simula 的面向对象

1967 年公布的 Simula 是世界上最早的面向对象语言。从 Simula 这个名字就可以看出它是一门建模（模拟）用的语言。这门语言是由奥利 – 约翰・达尔（Ole-Johan Dahl）和克利斯登・奈加特（Kristen Nygaard）二人开发的。他们在当时大学等机构中广泛使用的 Algol 的基础上引入了类和对象，以表达建模时用的实体（entity）。这里所说的实体指的是像交通系统建模中的信号灯和车辆这样的建模对象。

虽然 Simula 被称为世界上最早的面向对象语言，但这是现代人回顾过去得出的结论。“面向对象编程”这个词语被认为是由 Smalltalk 的开发者艾伦・凯（Alan Kay）“发明”的。从严格意义上来说，在 Simula 的那个时代，面向对象编程这个词语还不存在。

但是 Simula 从诞生之日起就具备了现在大多数面向对象语言所具备的功能，比如类、继承、对象、动态绑定、协程和垃圾回收等。也就是说，虽然当时还没有面向对象语言这个词语，但是在

① 原书名是『オブジェクト指向プログラミング』，目前暂无中文版。——译者注

这个词语出现之前面向对象编程的概念就已经存在了。可以说我们现在使用的面向对象语言都直接或者间接地受到了 Simula 的影响。在约五十年前，在面向对象这个词语还没出现时，面向对象语言 Simula 就已面世，从某种意义上可以说它是编程语言界的欧帕慈。[①]

开发者是和蔼可亲的人

达尔和奈加特凭借“通过设计编程语言 Simula 1 和 Simula 67，创造了面向对象编程的基本概念”这一成就，在 2001 年获得了计算机科学界的最高奖图灵奖。二人都于 2002 年去世。

在他们获得图灵奖两个月之前，我在丹麦 JAOO 会议上见到了奈加特教授。他是一位非常和蔼可亲的人，在座谈会上跟我聊了很多话题。

他笑着对我说：“什么？你在设计语言？那应该是面向对象语言吧。了不起！所有的面向对象语言都像是我的孙子一样，哈哈。”

Smalltalk 的面向对象

如果说 Simula 是面向对象语言的鼻祖，那最有名、影响力最大的面向对象语言当属 Smalltalk 了。不过现在的开发者们都没有直接接触过 Smalltalk，跟他们说起面向对象语言，他们可能最先想到的是 Java。

Smalltalk 是 20 世纪 70 年代初诞生于施乐帕克研究中心（Xerox Palo Alto Research Center，Xerox PARC）的一种面向对象语言。面向对象编程这个词语就是在 Smalltalk 的开发过程中诞生的。

Smalltalk 虽然受到了 Simula 的影响，但是它最主要的目标是成为 Dynabook[②]，即儿童也可以使用的未来计算机上的语言。

于是开发者们将重点放在了儿童也容易理解、可以直接操作的“对象”上，同时受到当时作为教学语言兴起的 LOGO 语言的影响，他们设计了以“通过向对象发送消息进行操作”这种模型为中心的语言。由“提笔”“前进 100 步”“向右转 120 度”等命令构成的 LOGO 的海龟绘图，在 Smalltalk 中也可以通过“海龟”对象和对海龟的命令这一模型实现。

用 Smalltalk 表达 LOGO

图 3-1 是用 LOGO 编写的海龟绘图的程序，图 3-2 是用 Smalltalk 编写的海龟绘图的程序。

---

① “欧帕慈”指的是当时那个年代的技术不可能加工出来的、颠覆常识的出土文物。

② Dynabook 是 Smalltalk 作者艾伦·凯在 1968 年提出的可以带着跑的电脑的概念，主要目标使用者是儿童，帮助儿童学习。——译者注

```
FORWARD 100
RIGHT 120
FORWARD 100
RIGHT 120
FORWARD 100
RIGHT 120
FORWARD 100
```

图 3-1　用 LOGO 编写的海龟绘图

```
Turtle go: 100.
Turtle turn: 120.
Turtle go: 100.
Turtle turn: 120.
Turtle go: 100.
Turtle turn: 120.
Turtle go: 100.
```

图 3-2　用 Smalltalk 编写的海龟绘图

这里需要简单说明一下 Smalltalk 的语法。`Turtle home` 部分的意思是“向 `Turtle` 对象发送 home 消息”。`Turtle` 对象响应该消息，把光标移动到 home 位置。

带参数的消息后面有“`:`”符号。比较特殊的是，当有多个参数时，需要在每个参数前带上消息。假设有一条消息可以在 `go` 的同时指定颜色，那么就需要在指定距离的同时指定颜色，这条消息的定义就会变成“`go:color:`”。实际调用此消息的代码大概如下所示，其中“`#red`”是 Smalltalk 的符号的写法。

```
obj go: 100 color: #red
```

这种写法看上去与其他语言的关键字参数很相似，但它既不能省略，也不能改变顺序。我们应该把“`go:color:`”看作一条分开写的消息。在 Smalltalk 中，包括控制结构在内的几乎所有的处理都是通过发送消息实现的。这也是 Smalltalk 的一个特征。

Ruby 与 Smalltalk 相似吗

按照发布年份，Smalltalk 有 Smalltalk-72、Smalltalk-76 和 Smalltalk-80 三个版本，现在说的 Smalltalk 就是指最后的版本 Smalltalk-80（及其派生版）。版本的每次进化都会让它愈发接近成人使用的语言和环境，为儿童设计的表情文字等功能则渐渐消失了。

多年前我与 Smalltalk 的开发者凯一起吃午饭，他对我说 Smalltalk 受到 Lisp 很大影响，已经跟当初设想的不一样了，还说 Ruby 和 Smalltalk-76 有一点像。Smalltalk-76 没怎么公开，资料也很少，所以我不是很清楚它和 Ruby 到底相像到何种程度，但他的这句话给我留下了深刻的印象。

Actor 的面向对象

受到 Simula 的面向建模以及 Smalltalk 的面向对象的影响，美国麻省理工学院的卡尔·休伊特（Carl Hewitt）在 1973 年建立了 Actor 模型。

在 Actor 模型中，各个对象独立进行计算，对象之间的通信通过消息进行。由于 Smalltalk 的消息发送需要等待结果返回，所以是“同步的”。而在 Actor 中对象间的通信则是异步的，只需发送消息即可，结果会以另一条消息的形式返回。

Actor 模型是基于并行计算机会在不久的将来出现这一预测诞生的。这种并行计算机由成百上千的微处理器组成，每个处理器都有自己的本地内存，通过高性能的通信网络进行通信。实际上在 1973 年那个时间点的“不久的将来”并没有出现这种计算机，所以 Actor 模型没能迅速普及，但是在 40 多年后的今天，由于多核、众核以及云计算的出现，休伊特的预测正在渐渐变为现实。

Erlang 也采用了 Actor 模型

提供 Actor 模型的语言在逐渐增加，比如 Erlang 就是以 Actor 模型为中心设计的语言。尽管 Actor 模型是受到面向对象思想的强烈影响提出来的，但 Erlang 的设计者乔・阿姆斯特朗（Joe Armstrong）却发表过面向对象没用的言论。不过时隔 40 年，他似乎又对休伊特的 Actor 模型有了新的发现，他说：“Erlang 的进程是对象，我发现 Erlang 才是真正的面向对象。”真是有趣！

CLOS 的面向对象

Lisp 是一门灵活度非常高的语言，非常适合用来测试编程中的新功能。面向对象编程也不例外，在 Simula 和 Smalltalk 发明出面向对象编程的概念之后，各式各样的面向对象系统就在各种 Lisp 语言处理器上进行过实验。这些面向对象系统中成就最大的当属 CommonLisp Object System（CLOS）。

CLOS 有如下特征。

- 多重继承（multiple inheritance）
- 实例方法（singular method）
- 多重方法（multimethod）
- 方法组合（method combination）

面向对象编程中的继承是指从现有的类继承功能，通过添加、修改功能来创建新的类。

包括 Ruby 和 Java 在内的许多语言只能从一个类继承功能，这种继承方式被称为单一继承（或者叫单重继承）。

多重继承是指不是从一个，而是从多个类继承。CLOS 和 C++ 支持多重继承。要继承的类从一个变成两个以上，可以说是比较自然的扩展，但实际上没有这么简单。继承两个以上的类时会出现很多问题，比如方法名冲突、类层次结构从简单的树结构变成网状结构等。Java 和 Ruby 为了避免这样的问题，采用了单一继承的方式。

而 CLOS 则设计了 Mix-in，引入了化解矛盾的结构。CLOS 的 Mix-in 像是多重继承使用方法上的“君子协定”，而 Ruby 则选择以模块的 include 的形式在语言层面进行特殊的处理。

“实例方法”不是指类层面，而是指为某个特定的对象定义的方法。CLOS 中用下面这样的代码代替参数的类名，就可以为对象（值）定义特有的方法（这里是 eql）。

```
(eql 值）
```

独立于类的方法

多重方法是属于多个类的方法。在很多面向对象语言中，方法属于类，通过类的对象来调用方法。可是在 CLOS 中，方法相当于函数，根据方法的所有参数所属的类来选择合适的方法。

我们通过一个实际的例子来加深理解。首先，CLOS 的方法调用看起来与普通的函数调用完全相同。

```
(length obj)
```

运行上面的代码，在属于名为 length 的函数（由于它可以代表多个方法，所以被称为广义函数）的多个方法中，匹配 obj 类的那个方法会被执行。如果用其他的面向对象语言进行调用，代码就会变成下面这样，可以看到顺序发生了变化。

```
obj.length()
```

可如果参数有多个，情况就不一样了，比如下面这行代码。

```
(plus obj1 obj2)
```

上面的代码可以调用拥有加法计算功能的广义函数，而具体执行哪个方法则由所有参数的类来决定。obj1 和 obj2 的类型是整数和浮点数的排列组合，每种组合都有各自的方法定义。这样就不需要根据参数的类型进行条件判断，可以选择更加合适的处理方式。这与 C++ 和 Java 的方法重载（overload）有些相似，不过多重方法的选择与普通的（根据第一个参数）方法选择一样，都是动态进行的。

这就发生了天翻地覆的变化——面向对象语言中常见的“先有类再有属于类的方法”这一结构完全不见了，取而代之的是“既有类又有独立于类的方法”这样的结构。虽然也有人质疑过这样是否还能叫面向对象，但是由于多重方法能够通过参数自动进行方法选择，而且与现有的 Lisp 的一致性也很高，所以 Lisp 界接受了这一结构。但是，除了 2015 年 12 月正式发布的 Perl 6，Lisp 之外的语言基本没有采用多重方法。

大规模的方法组合

CLOS 的最后一个特征是方法组合，即当有多个同名且可用的方法时，考虑如何组合这些方法进行调用。

许多面向对象语言都可以在方法中使用 super 等关键字来调用父类的方法，但是 CLOS 中由于多重继承的关系，不能以这种简单的方式来解决这个问题，于是 CLOS 干脆就允许自由定义方法的组合了。这实在是太灵活了。

举例来说，按照标准的方法组合方式，方法调用按照以下顺序进行。

首先，在属于被调用的广义函数的方法中，对匹配参数的可用方法按照优先级进行排序。优先级规则是：1 是“整数”，如果把“整数”当作“数”的子类，那么处理整数的方法要比处理数的方法优先级更高（Lisp 语法上“更加匹配”）。

其次，按照匹配度由高到低的顺序调用有“:around”标签的方法（如果存在的话），调用顺序如图 3-3 所示。在 around 方法内部，通过下面这行代码就可以根据优先级顺序调用下一个方法。如果不存在还未执行的 around 方法，则进入下一步。

```
(call-next-method)
```

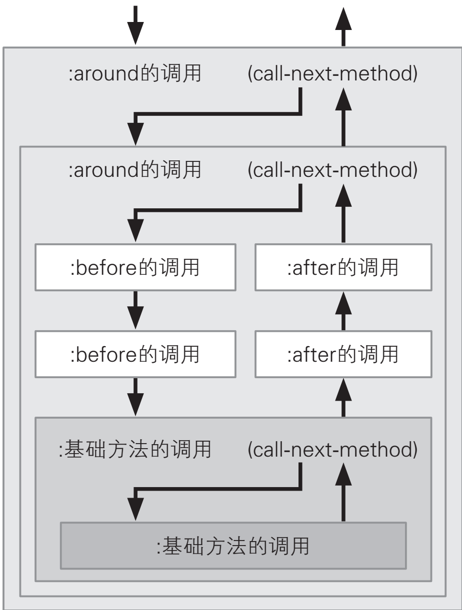


图 3-3　CLOS 的标准方法组合

接下来，按照匹配度由高到低的顺序调用有“:before”标签的方法（丢弃返回值），之后调用没有标签的匹配度最高的方法。与 around 方法一样，用 call-next-method 就可以调用后面匹配的方法。

最后，按匹配度从低到高的顺序开始运行有“:after”标签的方法（丢弃返回值）。

after 方法运行结束后，返回到 around 方法，然后一直执行到最后，这样最终的返回值就会是匹配度最高的 around 方法的返回值。

这种方法组合是面向切面编程的基础。实际上，Java 面向切面编程库 AspectJ 的开发者就是 CLOS 的设计者之一，即格雷戈尔・基克泽尔（Gregor Kiczale）。

Ruby 也借鉴了部分功能

老实说，CLOS 的面向对象功能非常多，在大部分情况下是超出需要的，我们基本上没有什么机会可以使用到全部功能。可是纵观面向对象编程的历史，我认为像 CLOS 这样经过深入研究

进行大胆设计的语言没有第二个。而且 CLOS 提供的那些未被其他面向对象语言引入的“独特的”功能，从某种意义上来说是“绝版”的技术，今后在设计面向对象语言时，有很多地方都值得借鉴。实际上 Ruby 提供的 Mix-in 与实例方法，以及 Module#prepend 等功能就是借鉴 CLOS 设计的。

虽然现在 CLOS 没有被广泛使用，但是这并不代表今后它不会流行起来。即使没有流行起来，CLOS 发明的功能也可能会被新的语言采用。

C++ 的面向对象

接下来要谈的是同为面向对象语言但背景却极为不同的 C++。C++ 在 C 的基础上增加了面向对象功能。很多面向对象语言都受到了 Smalltalk 的影响，但令人惊讶的是，在 C++ 身上却看不到这种影响。就拿术语来说，C++ 中父类不叫父类，叫“基类”（base class）；子类不叫子类，叫“派生类”（derived class）……由此就可以感受到其文化的不同。

我在 2004 年和 C++ 的设计者本贾尼・斯特劳斯特鲁普（Bjarne Stroustrup）进行过一次面对面的交流。2003 年在丹麦 JAOO 会议[①]期间我见过他，他对我说：“我的学生只知道 C++，不了解 Ruby，我想让你来给他们讲讲。”于是我在他任职的美国德州农工大学举办了讲座。

那时候我向他问起了 C++ 的起源，他回答我说：“我在英国剑桥大学读研究生的时候，写论文时需要进行建模。原本我想使用本科时使用的 Simula，但当时 Simula 速度太慢根本没法用，所以我就用了 BCPL（C 语言的前身）。后来到 AT&T 的贝尔实验室工作之后，为了实现能用的 Simula，我就开发了 C with Class 语言，这门语言后来就成为了 C++。”

也就是说，C++ 是没怎么受到 Smalltalk 影响的 Simula 的“直系子孙”，所以才故意不使用受到 Smalltalk 影响的术语。另外，很多面向对象语言会为了灵活性而牺牲性能，在这样的背景下，C++ 还是永远将性能放在第一位，这一点可能与斯特劳斯特鲁普有过“Simula 太慢”的体验有关。

当时的 C++ 没有异常机制，没有多重继承，也没有模版，是一门非常简单的语言，所以它作为静态类型的面向对象语言在实用性上略显不足。不过之后的进步让人刮目相看，现在它已经不单单是一门面向对象语言了，甚至可以说是一门有效使用模版的泛型语言，但也存在功能太多、较为复杂的缺点。

C++ 在以结构化编程为目标的 C 语言的基础上定义了面向对象，与用对象实体和消息发送模型

① 我参加过 2001 年和 2003 年的 JAOO 会议。前面提到过，在 2001 年的会议上我见到了奈加特教授，而且会议期间发生了“9・11”恐怖袭击事件，所以那次会议给我留下了很深刻的印象。另外在那次会议上我也第一次见到了为 Ruby 的普及做出卓越贡献的戴维・托马斯（Dave Thomas）。在 2003 年的会议上我见到了斯特劳斯特鲁普并受邀去他任职的大学举办讲座，还见到了 MVC 模型之父特里夫・雷因斯高（Trygve Reenskaug），这两件事让我印象深刻。

进行编程的 Smalltalk 可以说是一时瑜亮。我想过去的面向对象之争就是因为站在各自立场上的人没有充分理解对方才发生的。

Java 的面向对象

介绍完 Simula、Smalltalk、CLOS 和 C++，我觉得已经足够了，但这里还想再简单介绍一下其他有名的面向对象语言。

Java 是面向对象功能比 C++ 更接近 Smalltalk 一些的编程语言。它不像 C++ 那样特意避开 Smalltalk 的术语，还是按照一般的习惯来称呼父类和子类。

Java 作为面向对象语言的特征跟 C++ 很像，而且还积极引入了 Lisp 系语言常见的垃圾回收功能（基于性能上的考虑，C++ 没有采用这项功能）。另外，利用接口，Java 允许“事实上的多重继承”，不允许“实现上的多重继承”，给人一种严谨的印象。

Ruby 的面向对象

关于 Ruby 我也再顺便说几句。在设计思想层面 Ruby 和 Java 一样，都是处于 C++ 和 Smalltalk 之间的位置的语言，但是 Ruby 比 Java 更接近 Smalltalk。相比 C++，Ruby 的面向消息的色彩更加浓厚，比如 Ruby 的 `method_missing` 功能就是取自 Smalltalk 的 `DoesNotUnderstand` 消息的实现。

Ruby 不仅受到了 Smalltalk 的影响，还受到 CLOS 的影响，继承了实例方法和 Mix-in 等功能。不过 Ruby 没有引入 CLOS 的多重方法和方法组合等新颖且影响范围较大的功能。之所以这么做，是因为 Ruby 的目标不是成为实验用的语言，而是成为实用的面向对象语言。因此我采用了保守的设计方式，以避免给用户带来混乱。

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

时光机专栏

面向对象已经司空见惯了吗

本节是 2015 年 11 月刊中刊登的内容。我在本节根据历史从各种角度讲解了面向对象，完全没有涉及 Streem 语言。

面向对象的概念非常混乱，在还没有达成严格的定义之前很多人就开始了讨论，所以很容易发生偏离讨论主题、场面混乱等情况。面向对象是考虑问题的方法的指导，在各个语言之间会有微妙的差别，因此和背景不同的人谈不到一起也是情理之中的事情。

话虽如此，但讨论没有丝毫进展也是不行的，所以我觉得偶尔像这样从全局进行思考也是很有意义的。在本节中，我有意识地避免偏向某一门语言，尽可能地做到客观公正。虽然这些语言之中有一门是我设计的，平心而论，很难做到完全平等，但我依然以一个语言爱好者的身份编写了本节。

实际上，近些年已经没有关于面向对象的争论了，最近拿函数式编程和面向对象编程进行对比的事情反而多了起来。

我想可能是因为面向对象编程已经有些司空见惯了吧。关于结构化编程，几十年前也曾有过激烈的讨论，但现在它已经成为常识，所以也就没有人再争论了。同样的事情也正发生在面向对象编程的身上。作为一名资深的面向对象编程迷，我还真感到一丝寂寞。

3-2 Streem的面向对象

3-1节回顾了历史，了解了各种编程语言的面向对象功能，本节将开始设计Streem的面向对象功能。我们会实现动态绑定，根据实际数据的种类选择适当的处理。

马上就要开始设计 Streem 的面向对象功能了。不过 Streem 受到了函数式语言的强烈影响，如果把其他语言的面向对象功能原封不动地移植到 Streem 中，想必也不是很好用，所以我们先来复习一下 Streem 的特征，然后再思考什么样的面向对象功能才是最适合 Streem 的。

Streem 中需要动态绑定

Streem 最大的特征是大部分对象是不可变的，也就是不能更新的。大多数面向对象语言会通过修改对象的属性（实例变量等）来进行计算处理，也就是说，把状态封装在对象里，从而使状态更加容易处理。3-1 节介绍的最古老的面向对象语言 Simula 也是如此，为了管理模拟的状态而引入了对象，为了统一定义多个同种对象的行为而引入了类。

但是在受到函数式语言影响的 Streem 中，值在初始化之后就不能再被修改，也就是说，在 Streem 中根本不会发生“状态的管理”。既然没有状态管理，不会产生副作用，那 Streem 就没有什么必要去引入传统的面向对象了。

那么在 Streem 这样的语言中，面向对象编程完全没有用吗？

并非如此。在面向对象编程的多个特性中，动态绑定不会带来副作用，比较适合 Streem 这种语言。

动态绑定是指根据实际数据的种类选择适当的处理。比如，不管变量 `a` 的值是字符串还是数组，只要调用 `length` 方法，就可以求得它的长度。计算字符串长度的处理和计算数组长度的处理的内部实现完全不同，但是开发者不用在意这种内部实现的细节，只需在“计算长度”这一抽象层面考虑即可。如果要在 Streem 中引入面向对象功能，那么这个动态绑定功能是我首先想引入的。

广义函数

在引入动态绑定时，为了使其与 Streem 的类似于函数式语言的性质相匹配，我们首先引入广

义函数。广义函数（在 3-1 节也介绍过）是指 CommonLisp 等语言中采用的根据参数的类型选择内部处理的函数。比如在计算数据长度时，运行下面这行代码，就可以根据 a 的数据类型选择合适的计算长度的处理（方法）。

```
length(a)
```

CommonLisp 不仅根据第一个参数，还会根据所有的参数类型来选择方法。不过我决定先让 Streem 只根据第一个参数的类型来选择方法（偷个懒）。

广义函数可以在保持函数调用这一形式的基础上实现动态绑定。

在定义方法时，像下面这样指定参数的类型。如果名为 length 的广义函数不存在，就创建这个函数，并向这个广义函数注册处理 array 类型的方法。省略类型则意味着可以接受任意类型的数据。

```
def length(a:array) { ... }
```

由于方法定义本身可能会带来副作用，所以我就让 Streem 只能在顶层作用域（以及后面会介绍的命名空间）定义方法。像 Ruby 那样的带条件的方法定义是不允许出现在 Streem 中的。

减少类功能的增加所带来的副作用

Ruby 允许事后向类中添加功能。现在广泛使用的“猴子补丁”（monkey patch）技术就利用了这一特性来扩展现有类的功能。使用 Ruby on Rails 时常用的 ActiveSupport 库就是使用猴子补丁来向现有的类添加各种功能的。由此，我们就可以编写出下面这种与普通的 Ruby 代码截然不同的程序。

```
2.days.ago
```

猴子补丁虽然方便，但是也有副作用。如果事后毫无限制地向类中添加方法，那么名称相同功能不同的方法就可能会保存在多个库中。如此一来，在使用这样的两个库时，就可能会出现意想不到的问题。

为了避免这样的问题，Ruby 2.0 引入了 Refinement 功能。Refinement 的作用是只在某个特定的作用域内向类增加方法。

图 3-4 展示了 Refinement 的使用方法。

大家看明白了吗？在 using 指定的作用域之外，类没有变化，保持了原有的行为，但是在 using 指定的作用域内（从 using 出现到文件结束），Refinement 生效，增加的功能可见。这个切换是静态的，即使是同一个对象，在作用域内外也会做出不同的行为。这样我们就不需要担心名称

的冲突，以及方法增加所带来的副作用了。

不容易实现的 Refinement

尽管目前 Refinement 功能非常粗糙，将来还需要对其功能进行扩展，但今后它应该会取代猴子补丁。Refinement 功能目前是非主流的，所以很少有语言会提供类似的功能。只有 Java 中提供了进行类似的功能扩展的名为 Classbox 的处理器，还有某种 Lisp 方言中提供了类似功能的 Selector Namespace。

此外，Objective-C 的分类功能，以及 C# 和 Swift 的类扩展，也都与 Refinement 有些相似。

要实现 Refinement 非常困难。C# 和 Swift 可以使用静态类型的信息实现 Refinement，但在 Ruby 中一切都要在运行时解决，而且还不允许 Refinement 的实现使性能变低。从我说要在 Ruby 中实现 Refinement 功能到实际引入，总共花了将近十年的时间，在此期间我一直没有想到高效的实现方法。前田修吾也花了很长时间才帮我想出高效的实现方法。

广义函数与 Refinement

但是，令人意外的是，采用广义函数可以简单地实现与 Refinement 相同的效果。

广义函数从外表来看只是作为函数被处理。与 Refinement 相同的效果是指，可以根据作用域的不同使用同一个名称调用不同的函数。很多支持函数调用的语言已经以“作用域”或者“命名空间”为名实现了这项功能。

所以我打算在 Streem 中引入命名空间的概念，实现与 Refinement 相同的效果。Streem 的命名空间的语法如图 3-5 所示。

```
# 原来的类 只有foo方法
class Foo
  def foo
    p :foo
  end
end

# 成为Refinement单位的模块
module FooRefine
  # 对Foo类进行refine（扩展）
  refine Foo do
    # 扩展Foo类
    def foo
      p :foo_refine
      # 用super调用原来的方法
      super
    end

    # 增加新的方法
    def bar
      p :bar
    end
  end
end

# 创建Foo类的对象
f = Foo.new

# 调用Foo类的foo方法
f.foo   # => :foo

# Foo类中没有定义bar方法
# foo.bar  # Error! 因为方法不存在

# 用using使Refinement生效
using FooRefine

# 从这里开始扩展的foo与bar有效
f.foo   # => :foo_refine\n:foo
f.bar   # => :bar
```

图 3-4 Ruby 的 Refinement

```
namespace文    = namespace <名称> {
                   语句...
                 }
import文       = import <名称>
def文          = def <名称>(<名称>[:<名称>],...) {
                   <语句>...
                 }
名前           = [A-Za-z_][A-Za-z0-9_]*
```

图 3-5　Streem 的命名空间的语法

只看语法定义可能不太容易理解。图 3-6 展示了实际的程序示例，其中以注释的形式写下了想让程序实现的内容。

使用命名空间可以轻松地创建只能在某个特定作用域内可见的函数，这就意味着即使不引入 Refinement 这种很难实现的功能，也可以实现以下两个效果：定义只在某个作用域内有效的方法（Swift 和 C# 的类扩展所实现的效果）；临时修改限定在某个作用域内的方法的行为（Refinement 所实现的效果）（图 3-7）。

```
# 用namespace语句定义名为test_ns的命名空间
namespace test_ns {
  # 用import导入其他的命名空间
  import development_ns
  # development_ns提供的变量和函数可见

  # 函数定义
  def print(message) {
    puts(stderr, message)
  }
}
```

图 3-6　Streem 的命名空间的程序示例

```
# 全局函数foo
def foo(a) {
  # 把参数转换为字符串输出
  puts(to_s(a))
}

# 用来进行“类扩展”的命名空间
namespace extend {
  # 重写现有的函数
  def foo(a:string) {
    puts("foo\n")
    # 调用被重写的函数
    super(a)
  }
  # 只在该命名空间内有效的函数
  def bar(a) {
    puts("bar\n")
  }
}

import extend
foo("a")  # extend的foo被调用
bar("b")  # extend的bar被调用
```

图 3-7　使用广义函数和命名空间进行类扩展

名称冲突

使用 import 语句可以向某个命名空间导入其他命名空间的功能，但是当从多个命名空间进行 import 时，如果各个命名空间有同名的值，就会发生名称冲突。一旦发生冲突，就无法判断该使用哪个值，由此便会产生矛盾。

化解这个矛盾的办法有几种，其中最简单的办法是报错。也就是说，冲突的发生意味着功能方面重复较多，比较危险，本来就不应该组合使用。

另一种办法就是在 import 时通过重命名的方式避免冲突。这种做法确实容易理解，但同时也需要考虑很多细节，比如当原来的命名空间调用修改了名称的函数时该如何处理等，实现起来比较复杂。

另外还有当发生冲突时显式指定命名空间进行调用的办法。这种做法也许不会像重命名那样难实现，但也绝对说不上简单。

到底该如何处理，我也苦恼了很久。目前还是把简单和易于理解放在第一位，所以采用了冲突时报错的方式。之所以做出这样的决定，是因为从 Ruby 的经验来看，从多个命名空间（在 Ruby 中是模块）继承的名称发生冲突并且需要解决这个冲突的情况不是很多。但这并不等于今后不会采取报错以外的方式来解决名称冲突的问题，我就等需要解决时再考虑吧！

Streem 的对象

提起面向对象编程，就会想到定义类、创建对象、调用对象的方法等。

方法调用已经通过广义函数实现了，那么类和对象的相关内容该如何实现呢？前面也说过，我不想引入多个相似的数据结构，所以我想尽力避免引入新的“对象”类型。

这里我们参考一下其他语言。Lua 语言用表（table）这种数据结构替代对象。Perl 则是把散列当成对象使用。JavaScript 中虽然有对象这一数据类型，但本质上不过是散列表而已。

这就说明我们把现有的数据结构稍微包装一下就可以表示对象（或者是相当于对象的结构）了。Streem 里有数组，数组拥有一般语言中的散列表的功能，应该可以作为对象进行处理。

Lua 为了把表当成对象处理而设置了元表（metatable）。元表是保存对象的各种操作的表，可以说起到了类的作用。

Perl 提供了 bless 函数来构造对象，通过 bless 将标量类型的数据与包关联后，这个数据就可以作为对象使用。也就是说，对该对象的操作均由包中的函数负责执行。

我为 Streem 该采用哪种形式绞尽脑汁，苦恼了很久，最后从 Perl 中得到启示，决定把数组当成对象使用，也就是下面这种形式。

```
new <名称> [值...]
```

使用这种形式调用之后，会得到一个数组，这个数组把 < 名称 > 代表的命名空间当作类进行关联。

```
new Foo [1,2,3]
```

所以上面这行代码的意思是把命名空间 Foo 关联到拥有 3 个元素的数组并返回。通过对把数组当作第 1 个参数的函数调用进行特殊处理，就可以实现面向对象功能。

我想把命名空间当成类来使用，以实现面向对象功能。要想实现这个目标，就需要扩展函数调用的处理。扩展后的函数调用的处理如图 3-8 所示。当函数的第 1 个参数是用 new 创建的数组时，该命名空间的函数就会被调用。

这样就创建好了一个使用了广义函数和命名空间的比较简单的面向对象系统。

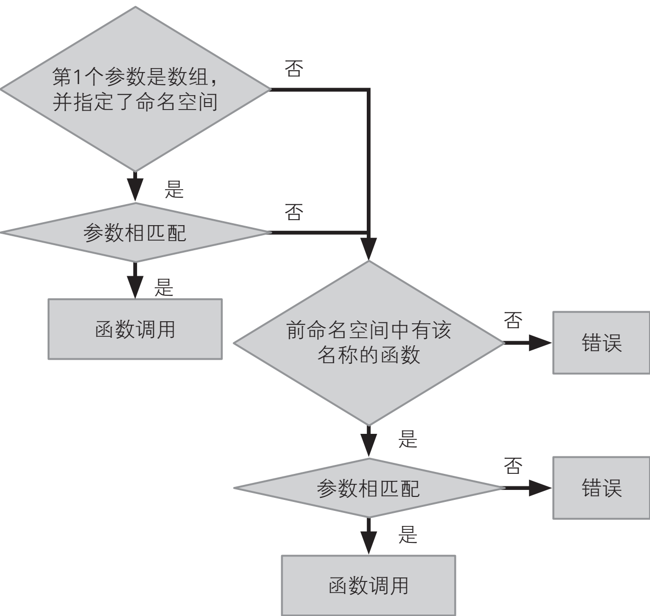


图 3-8 函数调用流程

“参数相匹配”是指参数的数量和类型与函数定义相符

方法调用链

不过函数风格并不总是合适的。连续调用多个函数时，函数调用的顺序和在程序中出现的顺序有时正好相反。

比如根据条件（数是偶数）筛选数组元素，对其排序并去掉重复元素，这一处理如果用函数调用的风格来写，就如图 3-9a 所示。想要实施的处理的顺序（筛选→排序→去重）与函数出现的顺序正好相反。

如果采用 Ruby 的方法调用风格编写这个处理，则如图 3-9b 所示。方法出现的顺序与实际的顺序一致，非常易于理解，而且这也与自然语言中“筛选、排序、去重”这一连续动作的表达方式一致。

其他语言大多也采用了类似的写法，有几种语言提供了连续进行函数调用的写法，比如 Elixir 的“|>”写法。图 3-10b 是这种写法的示例。

采用这种写法编写的程序就像是用方法调用风格编写的一样。因为这种写法非常方便，所以 Streem 中也决定引入。Streem 中用“.”代替了“$”，这样一来，写出来的代码看上去就像普通的面向对象语言的代码一样（图 3-10c）。

```
(a)函数风格
uniq(sort(filter(ary, {x -> x % 2 == 0})))

(b)方法风格
ary.filter{x-> x % 2 == 0}.sort.uniq
```

图 3-9　函数风格与方法风格

```
(a)函数风格
uniq(sort(filter(ary, {x -> x % 2 == 0})))

(b)$写法
ary $ filter{x-> x % 2 == 0} $ sort $ uniq

(c).写法（Streem采用的写法）
ary.filter{x-> x % 2 == 0}.sort.uniq
```

图 3-10　函数风格与连续进行函数调用的写法

Lisp-1 和 Lisp-2

在可以称为函数式语言鼻祖的 Lisp 中，函数的处理有两种方式，分别被称为 Lisp-1 和 Lisp-2。

Lisp-1 是 Schema 这个 Lisp 方言中使用的一种方式，函数的命名空间与变量的命名空间没有区别，命名空间只有一个（所以叫 Lisp-1）。在 Lisp-1 中，以下代码表示把 `"hello"` 当成参数传给 `print` 变量中保存的函数并执行。`print` 是对函数的引用。

```
(print "hello")
```

而 Lisp-2 是 CommonLisp 这个 Lisp 方言中使用的一种方式，函数的命名空间与变量的命名空间分离，所以会有两个（以上）的命名空间。在 Lisp-2 中，以下代码表示把 `"hello"` 当成参数传给 `print` 函数并执行。

```
(print "hello")
```

从表面上看代码与 Lisp-1 相同，但是进行 `print` 处理的函数即使引用 `print` 变量也无法取出，所以需要使用特殊的形式。CommonLisp 使用了下面这样的写法。

```
(function print)
```

或者使用下面这种省略的形式。

```
#`print
```

大家可能觉得 Lisp-1 和 Lisp-2 的区别微不足道，但其实它们都有各自的优缺点。Lisp-1 的优点是容易把函数作为值取出来，而且函数式编程的实现也比较简单。

在 Lisp-1 中，如下面这行代码所示，从返回函数的函数拿到返回值，向它传递参数并进行调用的处理很简单，因此使用 Lisp-1 也易于创建面向对象系统。但是 Lisp-2 就做不到这一点。

```
((func args) arg)
```

Lisp-1 的缺点是调用函数时必定伴随着对函数的引用，所以很难进行优化。Lisp-2 则正好相反。

除了 Lisp，其他语言也可以采用这种做法。实际上 JavaScript 和 Python 就采用了 Lisp-1 的方式，Ruby 和 Smalltalk 则采用了 Lisp-2 的方式。采用 Lisp-2 的方式是为了优化方法调用，但相应地把方法作为对象取出时会比较麻烦（图 3-11）。

```
# Python (Lisp-1)
obj.foo(1)              # 方法调用
f = obj.foo             # 方法取出
f(1)                    # 方法的间接调用

# Ruby (Lisp-2)
obj.foo(1)              # 方法调用
obj.foo                 # 方法调用（无参数）
f = obj.method(:foo)    # 方法取出
f.call(1)               # 方法的间接调用
```

图 3-11　Python 和 Ruby 的方法取出

需要注意的是，采用了 Lisp-2 方式的 Ruby 在取出方法时要使用 `method` 方法，在调用取出的方法对象时要使用 `call` 方法。

只看图 3-11 的代码可能会觉得 Python 的 Lisp-1 方式更加简洁，但 Ruby 采用的方式在方法调用的优化上有很大空间，还实现了方法缓存，而且还可以用类似于没有参数的属性引用的方式进行方法调用。

那么 Streem 该采用哪种方式呢？我打算采用 Lisp-1 和 Lisp-2 的中间方案，即 Lisp-1.5[①]。也就是说，在直接指定函数名时可以获取函数的引用，在使用了“.”的方法调用链中，可以将其看作没有参数的方法调用（表 3-1）。

表 3-1　Streem 的函数调用

写法示例	含义
`func`	函数 `func` 的引用
`func()`	函数 `func` 的调用
`a.b()`	函数 `b` 的调用（`b(a)`）
`a.b`	函数 `b` 的调用（`b(a)`）
`&func`	与 `func` 含义相同
`&func(1)`	函数 `func` 的偏应用函数
`&a.b`	函数 `b` 的偏应用函数（`&b(a)`）
`&a.b(c)`	`(&a.b)(c)`，即 `b(a, c)`

另外，使用“&”符号可以实现偏应用函数的效果。偏应用函数是指预先指定了一部分参数值

① Lisp-1.5 也是 Lisp 早期的一个语言处理器的名字，所以这里称为 Lisp-1.5 不是很妥当。

的函数。为了便于理解这个概念，我们来看一个实际的例子。

假设有一个对两个数值进行加法计算的 plus 函数。

```
plus(1,2) # => 3
```

上面的代码表示把 1 和 2 相加，返回 3。而运行下面这行代码，赋值给 plus5 的就会是“将 plus 的第一个参数固定为 5 的函数”。

```
plus5 = &plus(5)
```

向 plus5 传递参数，返回的就是该数值与 5 相加的结果。

```
plus5(1) # => 6
plus5(9) # => 14
```

这里只指定了一个参数，当然我们也可以指定两个以上的参数。

```
plus12 = &plus(1,2)
plus12() # => 3
plus12(3) # => ERROR! 参数过多
plus123 = &plus(1,2,3)
# ERRRO! 参数过多
```

这种形式对点（ . ）的写法也是有效的。f = &a.b 是 f = &b(a) 的省略写法，调用 f(c) 时 a.b(c)，也就是 b(a,c) 函数会被调用。

小结

本节介绍了 Streem 的面向对象功能的设计。Streem 从现有的语言中借鉴了很多东西，比如 CommonLisp 的广义函数和 Haskell 的“$”写法等，并巧妙地对它们进行了组合，从而设计出了风格独特的面向对象功能。另外，这部分内容也可以帮助读者认识到如何去进行语言设计。

本节的部分源代码在 https://github.com/matz/streem 上，这次的标签是 201512。不过由于实现的难度较大，所以使用“&”符号实现偏应用函数的功能等部分还没有着手开发，其余部分也还没有完成，仅供参考。

时光机专栏

应该有更好的办法

本节是 2015 年 12 月刊中刊登的内容。在 3-1 节介绍的面向对象的概念和历史的基础上，针对受到函数式语言影响的 Streem，本节探讨了如何设计面向对象功能。在面向对象编程成为常识的这个年代，最适合 Streem 这种语言的面向对象功能的设计方式却还没有一个正确答案。

这次尝试设计了使用广义函数和命名空间实现的面向对象功能，并开发出了能用的版本，这一点很不错，但还是有一些不尽如人意的地方。而且使用“&”符号的偏应用函数功能也没有开发完成。

我认为这次的设计比较好地融合了函数式编程和面向对象编程，但还是没能像 CLOS 的广义函数那样通过命名空间控制实现与 Refinement 同样的功能，所以还不能说完美地进行了融合。我心中还留有遗憾，希望将来能够做出更好的设计。

3-3 再看Streem的语法

经过一段时间之后再去看当初经过多方考虑设计的Streem语法，就会感到不满。趁现在还没有什么人使用Streem，正是改善语法的最好时机，所以我打算再来探讨一下Streem的语法。

写书时需要编写示例代码，所以我可能是世界上第一个用 Streem 写程序的人。我想至少现在除了我之外应该没人用 Streem 写程序。之所以这么说，是因为 Streem 还远远未达到实用的程度。

我在用 Streem 写程序时发现了一些不太中意的地方。在语言设计的过程中，发现当初设计时没考虑到的问题是常有的事。

shift/reduce conflict

首先是语法含混的问题。Streem 的语法是根据 yacc 工具的规则（yacc 定义文件）编写的，并使用 GNU 的 yacc 兼容工具 bison 编译为 C 代码。

把编写本节之前的 Streem 语法定义交给 bison 去分析，会得到下面的警告。

```
8 shift/reduce conflicts
```

这个警告的意思是“有 8 个语法含混的规则”。conflict（冲突）是指在语法分析的某个语境中，当某个记号来临时，存在多个应该迁移的状态。这种不知道语法规则的解析是该就此结束（reduce），还是继续进行（shift）的情况就称为 shift/reduce conflict。

就算当应该迁移的状态有多个时规则解析就此结束，也还是会出现下面这种错误。

```
reduce/reduce conflict
```

比如下面这个表达式。

```
expr + expr * expr
```

在这种情况下，并不能确定这个表达式应该解释为

```
(expr + expr) * expr
```

还是解释为

```
expr + (expr * expr)
```

所以 yacc 才会发出不知道该选哪个的 `shift/reduce conflict` 警告。但实际上，二元算术运算符是左结合的。

```
1 + 1 + 1
```

会被解释为

```
(1 + 1) + 1
```

另外，运算符有优先级，乘法运算符“`*`”比加法运算符“`+`”的优先级高，所以规则其实是很明确的。因此只要在 yacc 定义文件中告诉它规则，这个 `conflict` 就会消失。

bison 很智能，即使语法有些含混，也会帮我们生成可以适当进行解析的语法分析器。可如果语法有含混的地方，看的人也可能会被搞糊涂。

下面我们就去调查一下这次的 `conflict` 是在哪里发生的，是哪处语法不够明确。在执行 yacc 命令时添加 `-v` 选项，就会生成一个名为 y.output 的日志文件，用来记录语法是如何被解析的。

```
$ LANG=C yacc -v parse.y ↵
```

↵该符号代表回车

这里的 yacc 虽然是 bison 的别名，但若以 yacc 这个名字启动 bison，bison 就会以“yacc 兼容模式”运行。表 3-2 列举了 bison 模式下语法分析器的源代码和输出文件的名称。如果一个程序只处理一个语法分析器，我觉得还是 yacc 模式比较方便，所以就使用了 yacc 模式。

表 3-2　bison 模式与 yacc 兼容模式

模式	bison 模式[①]	yacc 兼容模式
语法分析器的源代码	parse.tab.c	y.tab.c
输出的日志文件	parse.output	y.output

为了防止输出的日志中出现英文以外的本地化字符，命令中指定了 `LANG=C`。虽然查看日志时中文更容易理解，但搜索日志内容时，“State”要比“状态”更容易输入，所以纯英文的日志反倒更加方便。当然这只是我个人的感想。

查看生成的 y.output 文件就会发现，文件前面显示了如图 3-12 所示的警告。这时如果查看

```
State  0 conflicts: 2 shift/reduce
State 11 conflicts: 2 shift/reduce
State 14 conflicts: 2 shift/reduce
State 51 conflicts: 2 shift/reduce
```

图 3-12　y.output 的警告

① 源代码文件是 parse.y 的情况。

State 11，就会看到如图 3-13 所示的内容。这里介绍一个小技巧，在查询状态时，使用 ^state 11 这样的正则表达式会比较方便。

通过图 3-13 我们可以得知是“\n”或者“;”导致了 conflict。实际上程序的结构是首先进行声明，然后执行语句，但由于这两部分的分隔符有“\n”和“;”，所以就出现了不知道该用哪个规则去解析的情况。当然，这只不过是分隔符而已，用哪个规则解析都不会有太大的区别，不会产生什么实质性的危害。但是放任不管总感觉不舒服，所以我还是决定修改一下规则。

```
State 11

    4 decls: decl_list . opt_terms
    6 decl_list: decl_list . terms
decl

    ';'   shift, and go to state 6
    '\n'  shift, and go to state 7

    ';'       [reduce using rule
106 (opt_terms)]
    '\n'      [reduce using rule
106 (opt_terms)]
    $default  reduce using rule
106 (opt_terms)

    opt_terms  go to state 50
    terms      go to state 51
    term       go to state 15
```

图 3-13　有 conflict 的 state

声明和执行语句

当前的 Streem 语法和以前的 C 语言语法一样，先进行声明，再执行语句，但其实并不需要这么做。于是我就让包括下列定义在内的声明成为顶级语句，使它们可以写在顶级作用域的任何地方。

- 方法定义
- 命名空间定义

这就意味着方法定义等不能写在条件表达式和方法定义之内，这样就达到目的了。

为新引入的顶级语句制定的规则（不包含动作）如图 3-14 所示。

```
topstmts : topstmt
         | topstmts terms topstmt
         ;

topstmt  : keyword_namespace identifier '{' topstmts '}'
         | keyword_class identifier '{' topstmts '}'
         | keyword_import identifier
         | keyword_method identifier '(' opt_f_args ')' '{' stmts '}'
         | keyword_method identifier '(' opt_f_args ')' '=' expr
         | stmt
         ;
```

图 3-14　顶级语句的规则

定义函数的 def 语句只是把函数对象赋值给变量，不是顶级语句，所以要把它放到普通语句那里。

另外，当 def 语句和 method 语句的定义部分只有一个表达式时，用“= 表达式”的形式来写即可。我本想把“=”也省略掉，但总是发生不明原因的冲突，所以我只好妥协，予以保留。

删除 break 语句

下面继续整理语法。当前语法中，中断函数运行的关键字有 break 和 skip 两个，两者作用相同，所以删除其中一个。break 最初是中断 C 语言循环的关键字，而 Streem 里没有循环，所以删除 break 关键字。

从词法分析器的 lex.l 和语法分析器的 parse.y 中删除 break 相关的部分即可。break 的相关处理还没有被运行过，所以删除了也不会出现任何问题。

我们顺便也让 skip 语句运行起来。当 skip 被调用时，让程序抛出异常，结束运行。

修改 if 语句

趁此机会，我们把对语法不满意的地方统一修改一下。

其中一个不满意的地方就是 if 语句。if 语句是 Streem 中为数不多的控制结构之一，可是当判断条件很多时，程序看起来就会很不美观。比如著名的 FizzBuzz 程序，用当前语法编写就如图 3-15 所示。

```
seq(100) | map{x ->
  if x % 15 == 0 {"FizzBuzz"}
  else if x % 3 == 0 {"Fizz"}
  else if x % 5 == 0 {"Buzz"}
  else {x}
} | stdout
```

图 3-15　用当前语法编写的 FizzBuzz 程序

让我不满意的地方有两点：一点是必须使用大括号“{ }”，另一点是没有用小括号围住条件表达式。

之前的语法要求禁止省略大括号是有原因的。在控制结构中使用大括号的语言，比如 C 语言，就发生过“dangling else”的问题。

这个问题是指，对于下面这段代码，

```
if (cond1)
  if (cond2)
    statement2
  else
    statement3
```

不知道是按照下面这种方式解释

```
if (cond1) {
  if (cond2)
   statement2
  else
   statement3
}
```

还是按照下面这种方式解释，从而产生了冲突。

```
if (cond1) {
  if (cond2)
   statement2
}
else
 statement3
```

即使产生冲突，yacc 也会以 `shift` 优先，按照“`else` 属于最近的那个 `if`”的规则去解释，但是程序依旧含混不清。

为了避免这个问题出现，Ruby 引入了“梳子型结构”。也就是说，只有一条语句时和有多条语句时的代码风格不同本来就是一个问题，所以不使用大括号，规定 `if` 语句到 `end` 结束。这样一来，整个 `if` 语句就会变成了下面这段代码的形式。`if` 和 `else` 等就像梳子齿一样凸出来，所以称为梳子型结构。

```
if cond1
  statement1
elsif cond2
  statement2
else
  statement3
end
```

而 Perl 则通过禁止省略大括号的方式解决了这个问题。之前 Streem 也参考了这个方式，规定不允许省略大括号。

但是 Streem 中常用的函数式编程经常会把 `if` 语句当作表达式来返回值。在这种情况下，作为表达式还是不使用大括号写起来更加自然。

所以我决定将 `if` 语句恢复为 C 语言的风格，将语法修改为单条语句（或单个表达式）时不使用大括号，包含多条语句时用大括号围起来。

与此同时，我也用小括号把条件表达式围起来了。之前的语法参考了 Go 的语法，没有使用小

括号围住条件表达式，但在 Streem 中，跟在函数调用之后的可能是代码块，如果不加小括号，语法就会变得非常复杂（为了在条件表达式中让代码块不跟在方法调用之后，我编写了两种独立的规则）。当时煞费苦心地编写了规则，但冷静之后再看就会觉得不惜让语法变得复杂而编写出来的规则并没有取得很好的效果，所以我想借此机会加以改善。

使用新语法编写的 FizzBuzz 程序如图 3-16 所示。虽然和图 3-15 的代码相差不大，但是看上去更加清爽了。

```
seq(100) | map{x->
  if (x % 15 == 0)      "FizzBuzz"
  else if (x % 3 == 0) "Fizz"
  else if (x % 5 == 0) "Buzz"
  else                  x
} | stdout
```

图 3-16　使用新语法编写的 FizzBuzz 程序

还有一些含混的地方

这次的修改虽然对机器来说规则明确无误，但却产生了让人容易弄错的组合，比如下面这行 `if` 语句。

```
if (cond) {print(x)}
```

其中，

```
{print(x)}
```

既可以看作用大括号围住了函数调用 `print(x)`，也可以看作一个定义部分是 `print(x)` 的函数对象，也就是省略了下面这段代码的参数部分的函数对象。

```
{->print(x)}
```

在这种情况下，Streem 将 `if` 语句的主体部分中出现的括号解释为用于围住代码语句的符号。

实际上这个实现非常麻烦，如果按照普通方式去写，语法分析器也会跟人一样出现混乱，从而发生冲突。于是，Streem 在实现时先把它当成函数对象进行语法解析，如果是 `if` 的主体部分，那就把它当成普通的大括号去重新构建语法树。

增加右侧赋值

不满意的地方还有很多。

在管道中赋值时让我感到特别不爽。具体来说，Streem 的管道由“|”连接，按照从左向右的方向进行处理。如果碰到分支处理等情况，就需要在管道中赋值，这时就要提前创建一个从左到右的管道，在需要赋值的时候返回到左边进行赋值，这正好与视线和思维的方向相反。

我们通过一个实际的例子来思考这个问题。图 3-17 是一个计算偏差值[①] 的程序（5-5 节会介绍）。

```
input = fread("result.csv")|map{x->number(x)}   # (c)
avs = input | average() # 平均值  (a)
sts = input | stdev()   # 标准差  (b)
zip(avs, sts) | each{x->
  avg = x(0); std = x(1)
  fread("result.csv") | map{x->number(x)}| each {score->
    ss = (score-avg)*10 / std + 50
    print("分数: ", score, "偏差值:", ss)
  }
}
```

图 3-17　偏差值的计算

这里需要注意的是图 3-17 的 (a) 部分。input 经过 average() 函数的计算，得到的平均值（的流）被赋给了 avs 变量。思维顺序明明是 input、average()、变量 avs，但是体现到程序上的顺序却是 avs、input、average()。(b) 部分也是如此。这种思维顺序和编写顺序的不一致会给人一种难以描述的负担，导致效率低下。

所以我决定使用“=>”符号从左向右赋值。这样一来，图 3-17 中的 (a) 就会变成下面这样的代码，思维方向与程序体现出来的顺序一致了。

```
input | average() => avs
```

编程语言一般都是从右向左赋值，也许有人会觉得执着于思维方向与出现顺序不一致的问题过于小题大做，但正是这种微不足道的改进的不断累积，才带来了语言易用性的提升。

修改函数调用

最后要修改的部分是函数调用。

前面也提到过，Streem 在设计上会尽可能地与 Ruby 不同。

另外，与一开始就作为面向对象语言设计的 Ruby 不同，Streem 更加重视函数式编程。因此，Ruby 以方法调用为中心，而 Streem 则把函数调用置于中心位置。

话虽如此，但我还是想提高面向对象的易用性，特别是想实现多态这项功能，所以引入了根据函数调用时第一个参数所属的命名空间来决定要调用的函数的结构。

---

① “偏差值”是指相对平均值的偏差数值，在日本常用于考察学生的智能和学力。计算公式为：偏差值 =（个人成绩 - 平均成绩）× 10/ 标准差 +50。——译者注

虽然每个设计都有它的理由，但是由此导致函数的定位发生了偏移也是不争的事实。

举例来说，如果是普通的函数式语言，下面这行代码的意思就是“用参数 x 调用名为 number 的函数（对象）”。

```
number(x)
```

而在 Streem 中，同样的一行代码却表示“参数 x 含有某个命名空间的信息，如果这个命名空间的作用域中存在名为 number 的函数，则调用该函数，否则将当前作用域中的名为 number 的对象当成函数来调用”。

```
number(x)
```

虽然函数调用的规则复杂了一些，但是熟悉之后其实并不难。麻烦的是在这种情况下，被调用的函数不能按照统一的名称进行处理。

请看图 3-17 的 (c) 部分。这行代码使用 map 把流中的元素转换为数值。map 会把函数当成参数处理，所以这里本可以用下面这行代码来调用。

```
map(number)
```

可是现在的 Streem 在进行 number(x) 函数调用（看上去像是）时，能够转换数据的是在各个命名空间定义的函数。通过 number 这个名称并不一定能够得到值（函数）的处理。因此无法像上面那样通过 number 这个名称取出“把各种数据转换为数值的函数”，也不能采用下面这种方式进行调用了。

```
map(number)
```

这样就不太好了，于是我决定引入在某个语境下与函数调用起相同作用的对象。由于分散在命名空间里的函数可以通过统一的名称处理，所以我把这种对象称为“广义函数对象”。

在标识符前加上“&”可以取出广义函数对象，因此下面这种写法

```
map(&number)
```

与前面的

```
map{x->number(x)}
```

作用相同。

函数的直接调用

另外，为了应对函数名和变量名碰巧产生冲突的情况，我采用了直接调用函数对象的方法。使用这个方法就可以无视多态，也不需要考虑第一个参数的命名空间。

```
func.(x)
```

像这样在参数的括号之前放一个“.”，就能保证函数对象被调用。与必须指定标识符的函数调用不同，在这种形式中，func 部分可以写成返回可以调用的对象的任意表达式。

Lisp-1 和 Lisp-2

Lisp 社区在很早以前就开始了这种关于函数调用形式的讨论。在 3-2 节也介绍过，Lisp 大体上可以分为 Lisp-1 和 Lisp-2 两种形式。

Lisp-1 是函数的命名空间和变量的命名空间不分离的形式。由于命名空间只有一个，所以叫 Lisp-1。

采用 Lisp-1 方式的语言有 Scheme、JavaScript 和 Python。这一设计方针也适用于 Lisp 之外的语言，但直到现在大家还是用 Lisp 的名字去称呼它，这让人感觉有点奇怪。

在采用 Lisp-1 的语言中，以下函数调用表示先评估 print 表达式，然后把参数 x 传给作为评估结果的函数（或者可以调用的对象）并进行调用。

```
print(x)
```

变量的引用相当于函数名，所以可以放置任何表达式。也就是说，

```
((complex_expr)(args1))(args2)
```

会进行以下动作。

- 评估 complex_expr
- 把 args1 作为参数传给作为上一步的执行结果的函数，并调用该函数
- 把 args2 作为参数传给作为上一步的执行结果的函数，并调用该函数

可以看出，通过很简单的规则也可以实现非常复杂的处理。

Lisp-1 结构简单，在其基础上还可以组建很多其他的结构，非常方便，所以追求简洁的 Lisp 方言 Schema 也采用了这种方式。

在 Lisp-1 中，函数对象的取出是基础操作，所以 Lisp-1 适用于频繁处理函数的函数式编程语言。最近出现的语言就经常使用这一形式。

Lisp-2 的优缺点

函数的命名空间和变量的命名空间相互分离的方式被称为 Lisp-2。采用 Lisp-2 的语言有 CommonLisp、Ruby 和 Smalltalk 等。

在采用 Lisp-2 的语言中，以下函数调用表示从函数表中搜索名为 `print` 的函数，然后把参数 `x` 传给该函数并进行调用。

```
print(x)
```

即使当前作用域中有名为 `print` 的变量，但由于变量的命名空间与函数的命名空间相互独立，所以也是毫无关系的。

Lisp-2 的“通过名称调用函数”的理念与面向对象的“通过消息调用功能（方法）”的理念非常相符，这就可以明白为什么原本作为面向对象语言设计的 Smalltalk 和 Ruby 会采用 Lisp-2 了。函数的搜索被完全隐藏在语言的运行时中，所以可以比较简单地实现方法缓存等能够提升函数调用性能的结构，这也是 Lisp-2 的一个优点。

Lisp-2 的缺点是不太擅长通过统一名称的方式处理应该被调用的函数，因此在进行函数式编程时会稍微麻烦一些。

同样的内容，如果用 Lisp-1 的方式来写，就是下面这种简单的形式，

```
map(number)
```

但是如果用 Lisp-2 来写，就得写成下面这种形式。

```
map({x->number(x)})
```

不过，如果引入前面介绍的广义函数对象，就可以简写到下面这种程度。

```
map(&number)
```

因此，我们可以认为这不是什么大的缺点。

Streem 和 Lisp-2

如前所述，最初在设计 Streem 时，考虑到函数式编程的便利性，我打算采用 Lisp-1 的方式。但由于我非常喜欢面向对象编程，总想以某种形式引入多态，所以在经过一番思考之后引入了 Lisp-2 的方式。

在这次改进中引入的广义函数使 Streem 能够在保持 Lisp-2 方式的同时采用类似于 Lisp-1 的方式来编写代码，我们也许应该称这种方式为 Lisp-1.5[①]。

小结

本节重新审视了 Streem 的语法，意外地发现了很多需要修改的地方。

现在 Streem 还没有用户使用，所以可以尽情修改。要是像 Ruby 那样拥有几万名用户，即使是微不足道的修改，一旦不兼容，也会成为大问题。

下一节我准备扩展 Streem 的语法，探讨一下函数式语言中常见的模式匹配功能。

时光机专栏

语言设计者如何改进语法

本节是 2016 年 8 月刊中刊登的内容。在连载即将结束时，我大面积修改了语法，本书根据最终的语法重新编写了示例代码并进行了替换。

所以本节介绍的改进语法的相关内容只能算是交代了修改语法的背景。2-4 节也有相似的介绍。我们可以看出随着连载的进行，语法也在一点一点地改善。

这里就体现出了连载的局限性。我不想大幅修改原来在杂志上连载的文章，所以这次就直接收录在书里了。希望本节内容可以让大家明白语言设计者在设计和改进语法时是如何考虑的。

通过本节我还感受到杂志连载的另一个局限性。3-2 节和 3-3 节的“Lisp-1 和 Lisp-2”的内容重复了。这本来就是我非常喜欢的话题，2015 年 12 月刊和 2016 年 8 月刊中都写到了这个内容。由于间隔了半年多，所以第二次写的时候完全没注意到这个话题重复了，而且当时也根本没有想到将来会把这个连载整合成书来出版。出书时根据内容重新编排了顺序，这时才发现内容重复了，也算是自食其果吧！

① Lisp-1.5 是 Lisp 早期的一个语言处理器的名字，这里只是一句玩笑话。

3-4 模式匹配

本节将介绍函数式语言中常见的一个功能——模式匹配。有了模式匹配，处理某些类型的数据结构就会变得非常容易。在经过反复试验之后，我终于在Streem中实现了这个功能。真想让Ruby也实现这个功能。

提起模式匹配，我们首先想到的就是正则表达式，函数式语言中的模式匹配起到的也是类似的作用，即检查值和模式是否匹配，以及从匹配的值中取出其中一部分。

Erlang 的模式匹配

不过在函数式语言中，匹配的对象不是字符串，而是数据结构。我们看一下具有模式匹配功能的函数式语言 Erlang 的例子。图 3-18 是一个不断求整数之和的斐波那契数列[①] 的程序。

```
fib(N) ->
  case N of
    1 -> 1;
    2 -> 2;
    _ -> fib(N-1) + fib(N-2)
  end.
```

图 3-18　Erlang 的模式匹配 (1)

Erlang 中使用 case 语句进行模式匹配（其实在函数定义中也可以进行模式匹配。这部分内容还请大家自行查阅 Erlang 的相关资料）。

case 语句把 N 代表的表达式与模式进行匹配，匹配成功后执行“->”之后的代码。如果模式中有变量，还会对这个变量进行赋值。

不过图 3-18 的例子太过简单，大家可能理解不了模式匹配与普通的条件判断有什么区别，又拥有什么特别的优点。我们再看一个稍微复杂一些的例子。图 3-19 是使用 Erlang 的模式匹配来定义计算链表长度的 len() 函数。

```
len(L) ->
  case L of
    [] -> 0;
    [_|T] -> 1 + len(T)
  end.
```

图 3-19　Erlang 的模式匹配 (2)

在图 3-19 的 case 语句中，最初的模式是 []，也就是空链表，这就意味着当表达式 L 与空链表匹配时，链表的长度为 0。

接下来的这个模式就有点特殊了。

---

① 斐波那契数列指的是这样一个数列：1, 1, 2, 3, 5, 8, 13, …。从第三项开始，每一项都等于前两项之和。本书的例子给出的数列第二项为 2，与一般的斐波那契数列略有区别。——译者注

```
[_|T]
```

这个模式的意思是链表最前面的元素为“_”，其余为 T，链表不为空时会匹配这个模式。在 Erlang 的模式匹配中，“_”这个变量可以匹配任意字符，这就表示它不关心匹配的内容，这样就可以把链表开头和末尾的元素逐个取出来。也就是说，去掉开头的一个元素之后，用 len() 计算链表 T 的长度，然后再加 1，即可求得这个链表的长度。这个程序运行后，会像图 3-20 一样进行计算。

```
len([1,2,3,4]) = len([1|[2,3,4]])
               = 1 + len([2|[3,4]])
               = 1 + 1 + len([3|[4]])
               = 1 + 1 + 1 + len([4|[]])
               = 1 + 1 + 1 + 1 + len([])
               = 1 + 1 + 1 + 1 + 0
               = 4
```

图 3-20 len 的计算

与递归组合使用会很方便

链表长度就是将开头元素的长度（1）加上剩余链表的长度，这种递归的定义也许不好理解，但熟悉之后就不会觉得难了。另外，在处理这种具有递归结构的数据结构时，使用模式匹配（与函数的递归调用的组合）会非常方便。这正是很多函数式语言具备模式匹配功能的原因。

除了这里介绍的 Erlang 之外，具备模式匹配功能的函数式语言还有很多，比如下面这几个。

- Haskell
- OCaml
- Scala

实际上，声称自己是函数式语言的编程语言基本上都具备模式匹配功能，不具备模式匹配功能的只有很久以前就存在的 Lisp（最近不怎么被当作函数式语言）及其直系子孙 Clojure 吧。

使用尾递归进行优化

顺便说一句，图 3-19 的递归函数在链表特别长的情况下会耗光调用栈。我们可以使用尾递归这项技术来解决这个问题。

尾递归是指在函数执行的最后调用自身。这种形式的递归函数调用可以优化为循环的方式，这样调用栈就不会被浪费，函数调用也可以省略。但如果是图 3-19 那样的结构，也就是下面这种结构，最后执行的就是“+”函数，即使进行了递归调用，也无法成为尾递归。

```
1 + Len(T)
```

如果像图 3-21 那样进行修改，就会成为尾递归。在 Erlang 中，参数个数如果不同就会被当成不同的函数，图 3-21 所示的程序就利用了这一点。如果是其他语言，只需修改一下有两个参数的那个函数名即可。这是将递归函数变为尾递归的一个简单的技巧。

```
len(L) -> Len(L, 0).
Len(L, Acc) ->
  case L of
    [] -> Acc;
    [_|T] -> len(T, Acc+1)
  end.
```

图 3-21　Erlang 的模式匹配（尾递归）

■ Streem 的模式匹配

接下来就要思考如何实现 Streem 的模式匹配了。

这里给大家介绍一件在我反复试验的过程中发生的事情。2016 年 5 月左右，函数的参数在某个瞬间由

```
{x -> print(x)}
```

变为了

```
{|x| print(x)}
```

这件事情的发生就是受到了反复试验的影响。我不慎提交了手头的修改，这个修改便向全世界公开了。结果在之后的东京 Ruby 会议的演讲中，我也不得不使用这个语法进行说明。

经过反复试验确定下来的 Streem 的模式匹配语法如下所示。

通过 case 和 if 语句实现模式匹配

首先，我们将模式写在函数对象的参数部分。要想放置模式，就需要在开头加上 `case` 关键字。

```
case 模式 -> 运行语句
```

另外，也可以像图 3-22 那样，通过并列多个 `case` 语句来指定多个模式。在所有模式都不匹配的情况下，可以使用 `else` 语句。

包含模式匹配的函数的调用与普通函数相同。如图 3-23 所示，函数一旦被调用，就会从上到下依次进行模式匹配，匹配成功时执行“`->`”之后的语句。

当有多个模式匹配时，只有第一个匹配到的模式生效。

```
foo = {
  case 1,x -> print(x+1)
  case 2,x -> print(x*2)
  else     -> print("else")
}
```

图 3-22　多个 case 与 else

如果所有模式均不匹配，则会抛出异常。

case 语句中也可以使用 if 条件语句（图 3-24），这种条件语句称为“守护”。如果不满足 if 指定的“守护”条件，即使与指定的模式相匹配，也会被认为匹配不成功。

```
foo(1,2) # => 3（根据2+1计算得出）
foo(2,1) # => 2（根据1*2计算得出）
foo(1)   # => else（不匹配）
```

图 3-23　模式匹配的执行示例

```
sign = {
  case 0 -> print("0")
  case x if x > 0 -> print("+")
  case x if x < 0 -> print("-")
}
sign(0)  # => "0"
sign(10) # => "+"
sign(-1) # => "-"
```

图 3-24　通过 if 添加“守护”表达式

通过 match 函数进行模式匹配

不进行函数调用，直接进行模式匹配时使用 match 函数。使用 match 函数的模式匹配如图 3-25 所示。match 函数在本质上只是对作为参数传过来的函数对象进行调用，但意外地与 Streem 的语法很相配，看上去像是其他语言的模式匹配的语法，而且作用也基本相同。

使用 Streem 的模式匹配语法重新编写如图 3-18 所示的程序，结果如图 3-26 所示。

```
match(1,2) {
  case 1,x -> print(x+1)
  case 2,x -> print(x*2)
  else     -> print("else")
}
```

图 3-25　match 函数

```
fib = {
  case 1 -> 1
  case 2 -> 2
  case n -> fib(n-1) + fib(n-2)
}
```

图 3-26　用 Streem 编写图 3-18 的模式匹配

■ 模式匹配的语法

前面粗略地介绍了模式匹配的相关内容，接下来我们考虑一下细节。

模式匹配的作用是把值与模式进行比较，判断值是否匹配模式。表 3-3 列举了模式的种类。

表 3-3　模式的种类

种类	示例	成功时的举动
变量	foo	与变量绑定
字符串	foo	匹配到字符串
数值	42	匹配到数值
数组	[1,a]	匹配数组的各个元素
结构体	[kw:a]	匹配结构体（带名字的数组）

变量模式

首先看一下第一个模式——变量。变量是 Streem 的标识符，以英文字符（字母）开头，后面

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

跟英文字符和数字。不过 Streem 的关键字不能是变量。变量可以匹配任意表达式。

变量分为“已绑定值的状态”和“未绑定值的状态”。通过赋值和模式匹配确定了值的变量称为“已绑定值的状态”，未确定值的变量则称为“未绑定值的状态”。对未绑定值的变量进行模式匹配时，会与尝试匹配的值进行绑定，通常会返回成功。对已绑定值的变量进行模式匹配时，如果尝试匹配的值与已绑定的值相等，则返回成功。

```
same = {
  case x,x -> true   # comparison in match
  case _,_ -> false  # fallback
}

print(same(1,1))  # => true
print(same(1,2))  # => false
print(same([1],[1])) # => true
print(same([1],[2])) # => false
```

图 3-27 使用模式匹配进行相等比较

我们通过一个例子来加深理解。图 3-27 所示为使用模式匹配进行相等比较的函数。

在 same 函数的模式匹配中变量 x 出现了两次，当这两个值相等时，结果为真（true）。

通配符模式

变量的匹配中有一个例外情况，那就是变量“_”永远不能变为已绑定值的状态。也就是说，同一个变量可以被匹配无数次。这样一来，在不关心变量的具体值的情况下，就可以使用变量“_”。由此，“_”称为占位符变量。由于“_”什么都可以匹配，所以有时候也叫作通配符变量。

图 3-27 的 same 函数的定义中出现了下面这行代码。

```
_,_
```

“_”与具体的变量不同，它什么都可以匹配，所以这里如果使用了“_”，那么即使第一个值和第二个值不同，也会被判断为匹配 。

“_”不能变为已绑定值的状态就意味着无法取出值，即使匹配也不能使用变量“_”的值。这种情况会按照未定义的变量进行处理，即报错。

字面量模式

如果仅仅是变量的罗列，那么模式匹配与一般的函数参数也没有多大区别。两者的不同之处就在于模式匹配除了变量，还可以直接罗列字面量（literal）。

字面量是指具体的值。如果模式中有字符串和数值，那么在与值本身匹配时，则匹配成功。Streem 的模式匹配中出现的字面量有字符串、数值、true、false 和 nil。

由于数组把模式包含在元素之中，所以与字面量的处理方式不同。

数组模式

数组模式用于匹配数组，其基本形式如下所示，中括号“[]”之间排列的是用逗号分隔的模式。

```
[ 模式 , 模式 ,...]
```

数组匹配在满足下列条件时匹配成功。

1. 要匹配的值是数组
2. 模式的数量与数组元素的数量相同
3. 所有的模式匹配均成功

数组模式匹配的例子如图 3-28 所示。

```
match([1,2,3]) {                    # 对数组[1,2,3]进行匹配
  case [a]        -> print(1)       # 与1个元素的数组匹配
  case [a,b]      -> print(2)       # 与2个元素的数组匹配
  case [1.5,b,c]  -> print(1.5)     # 与开头元素是1.5的3个元素的数组匹配
  case [a,"b",c]  -> print("b")     # 与第2个元素是“b”的3个元素的数组匹配
  case [a,b,c]    -> print(3)       # 与任意的3个元素的数组匹配
  case _          -> print("any")   # 不论是否是数组，都进行匹配
}
```

图 3-28 数组模式匹配

由于数组的匹配是递归结构，所以也可以写出数组的数组等模式，比如下面这个模式。

```
[[a,b],[c,d]]
```

这个模式中的 2 个元素分别是 2 个数组，每个数组（元素）中又包含 2 个元素。如果匹配成功，各个元素的值会与名为 a、b、c、d 的变量绑定。

如果把这个模式修改为

```
[[a,b],[a,b]]
```

那么只有 2 个元素相等的数组才会匹配，例如下面这个数组。

```
[[1,2],[1,2]]
```

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

可变长度数组模式

除了把模式作为数组元素的数组模式，Streem 还提供了从长度不明确的数组中取出其中一部分元素的方法。在这种数组元素的模式中，添加“* 变量”之后，其余不匹配的元素就能够作为数组与变量绑定，如图 3-29 所示。

```
match([1,2,3]) {
  case [a,*b] ->
    print(a)   # => 1 （第1个元素）
    print(b)   # => [2,3] （其余的元素）
}
```

图 3-29　可变长度数组模式

可变长度数组“* 变量”只能在数组模式的任意位置出现一次。也就是说，它既可以像前面的例子一样出现在末尾，也可以像下面这样出现在开头。

```
[*a,b]
```

当然也可以出现在中间，如下所示。

```
[a,*b,c]
```

但如果是像下面这样在同一层级中出现了多次可变长度数组模式，那么匹配的范围就无法确定，这种情况就会被当成语法错误来处理。

```
[a,*b,c,*d]
```

本节开头出现的 Erlang 的模式，即

```
[H|T]
```

在 Streem 中为

```
[H,*T]
```

其实 Streem 的内部实现是数组，Erlang 的内部实现是链表。虽然二者的内部实现不同，但是匹配开头的元素和剩余元素这一动作是相同的。

结构体模式

在 Streem 中，数组的各个元素都会被赋予名字，这就是所谓的“带标签的数组”，这里称为结构体。

结构体也是模式匹配的对象。结构体模式的语法如下所示。

```
[标签：模式， ...]
```

当指定标签的模式与其相对应的值全部匹配时，判定为结构体整体匹配。

基本规则很简单，但实现时我们也要考虑到极端情况。

最先需要思考的一点是，由于结构体的内部实现是带标签的数组，所以元素之间会有明确的顺序。我们需要决定在模式指定的标签与数组的标签的顺序不同的情况下是否应该匹配。

从易用性的层面来看，还是不考虑顺序为好。作为结构体使用的数据结构很少将各个元素的顺序纳入考虑范围内，更何况正是由于不想考虑顺序才添加的标签。

接下来需要思考的是，为了匹配结构体，是所有的标签都要一致，还是只要部分一致即可认为匹配成功。

我们也尝试从易用性的角度来思考这一点。

如果强制完全匹配，那么向结构体中添加元素时，所有的模式都需要进行相应的修改。如果允许部分匹配，那么从构造体中取出部分元素时就可以使用模式匹配。这样看来，允许部分匹配会更好。

最后需要思考的是，构造体的标签其实是允许重复的，那么当同一个构造体中有多个同样的标签时，该如何进行模式匹配呢？比如下面这个结构体。在这种情况下，匹配标签 `a` 的模式应该匹配 `1` 还是匹配 `3` 呢？

```
[a:1,b:2,a:3]
```

对此，我想到了两种解决方案。

1. 总是匹配最前面的标签
2. 按照标签的出现顺序进行匹配

方案 1 非常简洁。它的优点是实现起来比较简单，而且行为也容易预测，但是只能匹配到多个值中的第一个值。

如果是下面这种方案 2 的模式，带有标签 `a` 的第一个模式就会匹配到第一个 `a` 的值（`1`），带有标签 `a` 的第二个模式就会匹配到第二个 `a` 的值（`3`）。

```
[a:x,a:y]
```

方案 2 更加全面，所以用户体验会更好，但这次我决定采用较为简单的方案 1。因为从过去的经验来看，方案 2 很可能会带来意想不到的问题。在这种异常情况上花费过多的精力有些得不偿失，而且行为难以预测还可能会给用户带来困扰。

图 3-30 是最终确定的结构体模式匹配的示例。使用这个结构体模式匹配，就可以轻松地从 CSV 读取到的数据中抽取一部分并显示，具体代码如图 3-31 所示。

```
match([a:1,b:2,c:3,a:4]) { # 结构体（带标签的数组）模式匹配
 case [a:a, c:c, a:x] ->   # 即使不使用b也可以匹配
                           # 由于只使用第一个标签a，所以x匹配到的值也是1
                           # 不考虑标签的顺序
    print([a,c,x])         # => [1,3,1]
}
```

图 3-30　结构体模式匹配

```
# 有以下CSV文件（voters.csv）
# name,address,age
# 松本行弘，岛根县松江市，51
# 松本拓人，岛根县松江市，19
# 松本××，岛根县松江市，4
# 松本××，岛根县松江市，1

# 从voters.csv中筛选并显示有选举权的人
fread("voters.csv")|csv()|each{
  # 从2016年开始18岁以上的人拥有选举权①
  case name:name, age:age if age>=18 ->
    print(name)  # 显示有选举权的人的名字
  else ->        # 没有选举权就什么也不做
}
```

图 3-31　CSV 的读取和数据抽取

结构体模式也可以应用于可变长度数组。不过由于结构体模式不考虑元素的顺序，所以“* 变量”只能出现在末尾。这时会把已经指定了标签的值以外的值作为一个结构体（带标签的数组）赋值给“* 变量”。

我们来看下面这个示例。

```
match([a:1,b:2,c:3]) {
  case [a:x,*z] -> print(x,z)
}
```

这段代码会输出什么呢？答案如下所示。

① 这里是指日本的情况。——译者注

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

```
1 [b:2,c:3]
```

也就是说，根据 a:x 模式，标签 a 所对应的值 1 会赋值给变量 x，而标签 a 以外的元素会作为结构体赋值给 z。

那么从结构体中取出可变长度数组时，如果存在多个同名的标签，会返回什么值呢？我们分别从指定了重复的标签和指定了非重复的标签这两种情况来考虑。

下面的程序会分别输出什么值呢？

```
指定了重复的标签的情况
match([a:1,b:2,a:3]) {
  case [a:x,*z] -> print(z)
}

指定了非重复的标签的情况
match([a:1,b:2,a:3]) {
  case [b:x,*z] -> print(z)
}
```

大家不妨利用这个机会，在纸上列出可能实现的模式，考虑一下哪种做法更好。易用性和实现的复杂程度等都是需要考虑的因素。

大家考虑好之后，可以下载最新的 Streem 代码并编译执行上面的代码，看看会返回什么结果，比较一下是否与自己的结论相同。

假如有读者读了这一节，并且思考了这个问题，那么这些读者所做的事情正是语言设计者日常进行的语言设计活动。

我希望读者中会有人想在今后去尝试设计语言。

命名空间模式

我们在 3-2 节介绍过，Streem 中所有的值都带有命名空间，命名空间汇总了该值可以使用的函数。Streem 的命名空间就相当于其他语言中的类，用来表示值的类型。

在确认类型时可以使用命名空间模式。命名空间模式的表示方法是在其他模式之后加上“@ 命名空间名”。

比如下面这个模式在值的命名空间为 string 时则匹配成功，将字符串赋值给 str 变量。

```
str@string
```

命名空间模式还有另一种表示方法，如下所示。

```
[@ 命名空间名 值 , ...]
```

只需在数组（或者结构体）的开头加上“@ 命名空间名”即可，含义与下面这种写法无异。

```
[ 值 , ...]@ 命名空间名
```

这是在强调该数组是使用了命名空间的特殊的数组时使用的表示方法。

请记住用

```
new namespace [ 数组 ]
```

创建的对象可以用以下模式去匹配。

```
[@namespace 数组模式 ]
```

小结

本节讲解了 Streem 中新引入的模式匹配功能，同时也向大家介绍了在设计模式匹配行为的过程中，语言设计者是如何进行思考的。

时光机专栏

最适合Streem的模式匹配功能

本节是 2016 年 9 月刊中刊登的内容。这次设计并实现了很多函数式语言都有的模式匹配功能。模式匹配功能还是很有必要的，特别是根据不同的模式来分别处理管道中的数据这一做法，非常适合 Streem 所推崇的流编程。我非常期待模式匹配在 Streem 编程中的表现。

其实我也很想在 Ruby 中引入模式匹配功能，但由于这个功能与现有语法相冲突，再加上有功能上的一些限制，所以目前还没有实现。不过过去也有一些功能原本在 Ruby 中不存在，后来经过反复试验而最终实现了的（比如 Refinement），所以我希望将来也能实现模式匹配。

杂志连载的顺序与实际成书时的顺序不同，连载时本节内容还包括了 CSV 读取功能，但这部分内容放到了本书的 5-3 节。

第 4 章

实现Streem的对象

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

4-1 套接字编程

通过前面的开发，Streem语言已经达到姑且可以使用的程度了。本节先暂停语言处理器的话题，来介绍一下网络编程的相关内容，在Streem中增加使用套接字进行网络通信的功能。

如今，网络就像空气一样不可或缺。最近计算机作为单机使用的情况越来越少，很多应用必须连接网络才能使用。出差坐飞机时不能联网，更加让我意识到网络对于日常使用的应用来说多么重要。

虽说并不是必需的，不过我们还是给 Streem 加上网络通信的功能为好。这也是 Streem 功能扩展的一个好例子。

Streem 的套接字 API

首先来看一下用 Streem 编写的使用套接字的程序。图 4-1 所示为用 Streem 开发的网络服务器的代码，图 4-2 所示为客户端的实现。

这些代码都过于简单，让人有些提不起劲来。接下来我就逐行解释一下。

```
01 # simple echo server on port 8007
02 tcp_server(8007) | {s -> s | s}
```

图 4-1　用 Streem 开发的网络服务器的代码

```
01 s = tcp_socket("localhost", 8007)
02 stdin | s
03 s | stdout
```

图 4-2　用 Streem 开发的网络客户端的代码

Streem 网络服务器

我们先看图 4-1 的代码。第 1 行只是注释，意思是“在 8007 端口监听的 echo 服务器”。整个程序只有 2 行，但实际上只用了 1 行就写出了网络服务器的代码。

网络连接通过主机名和端口（端口号或服务名）指定。这个服务器启动后，客户端可以通过指定的主机名和端口号（8007）连接。

`tcp_server(8007)` 是创建服务器的函数。这个函数的作用是创建服务器套接字，等待客户

端连接，当接收到客户端连接时，创建相应的客户端套接字，并向管道发送数据。

接下来的 {s -> s | s} 是函数的主体部分。在 Streem 中函数用管道连接，把流向管道的各个元素作为参数传给函数，并调用该函数。示例中的函数把连接客户端的套接字作为参数，通过管道将其连接。乍一看也许不知道 s | s 是用来做什么的，其实它的意思是“将收到的客户端套接字的输入直接返回”。请注意套接字是可以双向通信的，既可以接收数据，也可以发送数据。

普通的服务器会把读取到的信息加工之后再返回，处理会更加复杂一些。

Streem 网络编程的客户端

图 4-2 的客户端程序也很简单。第 1 行代码 tcp_socket("localhost", 8007) 会生成连接 localhost 主机的 8007 端口的套接字，并赋值给变量 s。

第 2 行表示把从标准输入（stdin）接收到的数据发送给套接字，这样就可以通过套接字向网络服务器发送数据了。

第 3 行表示把从网络服务器接收到的数据发送给标准输出（stdout）。

这个程序与图 4-1 的 echo 服务器建立连接后，双方就会进行网络通信，把从键盘输入的字符串原封不动地返回（如果不考虑网络通信，这其实就相当于执行没有参数的 cat 命令）。

可以看出，即使是连接了套接字的网络编程，也可以使用 Streem 轻松地编写。

Streem 的功能扩展

接下来我们看一下 Streem 如何实现这种套接字功能。首先从向 Streem 添加函数开始。

从图 4-1 和图 4-2 的程序中我们可以看出，套接字功能是通过在 Streem 中增加 2 个函数实现的。向 Streem 添加函数定义时，代码如图 4-3 所示（为了方便说明，我对实际代码做了一些修改）。这次解说的实际的 C 代码保存在 Streem 源代码的 socket.c 文件中。

```
int tcp_server(strm_state*, int, strm_value*, strm_value*);
int tcp_socket(strm_state*, int, strm_value*, strm_value*);

void
strm_socket_init(strm_state* state)
{
  strm_var_def("tcp_server", strm_cfunc_value(tcp_server));
  strm_var_def("tcp_socket", strm_cfunc_value(tcp_socket));
}
```

图 4-3　向 Streem 中增加函数定义

在 Streem 中，用 C 开发的（全局）函数是按照以下步骤定义的。

首先从 C 函数指针开始创建函数对象。这里使用的是 strm_cfunc_value() 函数。用 strm_cfunc_value() 添加的函数的返回值类型必须是 int，而且要有 4 个参数：第 1 个参数 strm_state* 是方法调用的语境；第 2 个参数 int 是传给 Streem 函数的参数的数量；第 3 个参数是保存参数的数组；第 4 个参数是保存返回值的地方。函数本身的返回值代表函数的执行结果，成功时返回 STRM_OK(=0)，失败时返回 STRM_NG(=1)。

接下来将这些函数对象赋值给全局变量。定义全局变量时使用的是 strm_var_def() 函数。

用于初始化的函数 strm_socket_init() 由解释器的初始化函数 node_init() 调用。增加新功能之后，不要忘了从 node_init() 调用新功能的初始化函数。

什么是套接字

那么类 UNIX 操作系统中的套接字到底是什么呢？

套接字是用来作为网络连接通道的“操作系统对象”。据说套接字是在 BSD4.2 系统中被发明出来的。

套接字通过被称为文件描述符的整数来识别。这种整数与在 open 系统调用中打开的文件等是相同的标识符。此外，套接字也支持对文件描述符实施一些通用的操作，比如可以进行 read（读取）、write（写入）、select（等候）、close（结束）等处理。

不过仔细想一想，代表磁盘文件的文件描述符与代表网络连接的套接字的文件描述符在读写等处理上不可能完全相同。这就说明系统调用在内部会根据文件描述符的种类自动选择合适的处理。

根据对象的种类自动选择合适的处理也是面向对象的一种形式。从这个角度来看，UNIX 其实是面向对象操作系统，真是令人惊讶。

客户端套接字

套接字的使用方法并不难，但是在使用 C 语言进行套接字编程时，（可以非常细致地进行设置的）步骤有很多。这里我们对大致步骤进行说明，首先从相对简单的客户端开始。

从客户端开始的套接字通信步骤如下所示。

1. 获取要连接的服务器的信息
2. 生成套接字
3. 连接
4. 输入输出

首先指定要连接的服务器。网络服务器通过组合主机名和端口来指定。TCP/IP 连接时，主机用

IP 地址表示。我们平常看到的“192.168.0.1”“127.0.0.1”这种 4 个（255 以下的）数字的组合就是 IP 地址。起初 TCP/IP 地址用 32 位（4 字节）表示，IP 地址则是将各字节转换为十进制来表示。

但是现在 IPv4 地址濒临枯竭，于是便出现了使用 128 位的 IPv6 地址。

在 IPv4 地址还是主流的时候，从主机名获取 IP 地址时使用的是 gethostbyname() 函数，但现在通常都会使用可以区分 IPv4 和 IPv6 地址的 getaddrinfo() 函数。

使用 getaddrinfo() 函数可以获得地址、端口和套接字类型等信息。之后使用这些信息，用 socket() 系统调用生成套接字，然后用 connect() 系统调用连接服务器即可。

连接之后套接字会作为普通的文件描述符使用，因此可以用 read() 和 write() 函数进行数据的读写。除了这两个函数之外，还有 recv（读取）和 send（写入）等其他读写套接字数据的系统调用，不过目前我们还用不到它们，之后有机会再进行介绍。

套接字的使用方法（客户端）

下面我们就一边阅读 Streem 实现的套接字功能的代码，一边看一下如何用 C 进行套接字编程。图 4-4 是 Streem 的 tcp_socket 函数的实现。为了方便说明，这里删除了 Win32 相关的代码。

```
static int
tcp_socket(strm_state* state, int argc, strm_value* args, strm_value *ret)
{
  struct addrinfo hints;
  struct addrinfo *result, *rp;
  int sock, s;
  const char *service;
  char buf[12];
  strm_string* host;

  if (argc != 2) {
    return STRM_NG;
  }
  host = strm_value_str(args[0]);
  if (strm_int_p(args[1])) {
    /* 指定字符串形式的端口号 */
    sprintf(buf, "%d", (int)strm_value_int(args[1]));
    service = buf;
  }
  else {
    strm_string* str = strm_value_str(args[1]);
    service = str->ptr;
  }
```

```
  memset(&hints, 0, sizeof(struct addrinfo));
  hints.ai_family = AF_UNSPEC;    /* Allow IPv4 or IPv6 */ ◀— 设置hints的值
  hints.ai_socktype = SOCK_STREAM;
  s = getaddrinfo(host->ptr, service, &hints, &result); ◀— getaddrinfo()函数

  if (s != 0) { /* getaddrinfo失败 */
    /* 用gai_strerror()调查错误的原因 */
    node_raise(state, gai_strerror(s));
    return STRM_NG;
  }

  /* 遍历addrinfo尝试连接 */
  for (rp = result; rp != NULL; rp = rp->ai_next) {    ▼ socket()函数
    sock = socket(rp->ai_family, rp->ai_socktype, rp->ai_protocol);
    if (sock == -1) continue; /* 如果失败则尝试下一个 */
    /* 尝试连接 */
    if (connect(sock, rp->ai_addr, rp->ai_addrlen) != -1) ◀— connect()函数
      break;                    /* 如果成功则跳出循环 */
    close(sock); /* 关闭套接字再次尝试*/
  }
  /* 释放addrinfo结果的内存 */
  freeaddrinfo(result);

  if (rp == NULL) { /* 连接全部失败 */
    node_raise(state, "socket error: connect");
    return STRM_NG;
  }
  /* 把socket封装到strm_io对象并返回 */
  *ret = strm_ptr_value(strm_io_new(sock, STRM_IO_READ|STRM_IO_WRITE|STRM_IO_
FLUSH));
  return STRM_OK;
}
```

图 4-4 tcp_socket 函数

第 1 步是获取要连接的服务器的信息。这里使用的是 getaddrinfo() 函数，getaddrinfo() 将获取 host、service、hints 和 result 这 4 个参数。

向第 1 个参数 host 传递的值是主机名。这时可以传递 "127.0.0.1"（IPv4）和 "::1"（IPv6）等表示 IP 地址的字符串。

向 service 参数传递的值是服务名。这时系统会到“/etc/services”文件中根据服务名查找端口号。当传递的值是 "8007" 这种由数字组成的字符串时，就把这个值当作端口号使用。主机和服务可以有一个为 NULL，但是如果两个都为 NULL，系统就无法获取信息，从而报错。

向 hints 参数传递的值是作为获取信息的选项的 addrinfo 结构体。在没有什么需要指定的情况下，传递 NULL 即可。这次由于 IPv4 和 IPv6 的地址都可以使用，所以将 ai_family 设为 AF_UNSPEC。另外，由于搜索的是 TCP 连接而非 UDP 连接，所以将 ai_socktype 设为 SOCK_STREAM。

sockaddinfo() 在成功获取数据时会返回除 0 以外的值。如果因为某种情况而获取失败，可以用 gai_strerror() 函数得到表示错误原因的字符串。

getaddrinfo() 成功时，从传递了指针的变量中可以拿到 addrinfo 结构体形式的执行结果。当有多个查询结果时，addrinfo 结构体会以链表的形式存在，可以从开头按顺序尝试。

作为结果返回的 addrinfo 结构体在使用后需要用 freeaddrinfo() 函数进行内存释放。

获取信息之后，套接字连接就很简单了。使用 socket() 系统调用创建套接字，再用 connect() 系统调用连接服务器。调用 socket() 和 connect() 时需要的信息都可以在 addrinfo 结构体中找到。

connect 系统调用成功后，就建立起了与服务器的连接。之后就与普通的文件描述符一样，可以使用 read 和 write 进行读写。需要注意的是，套接字可以使用同一个文件描述符进行数据的读取和写入，这种可以双向通信的特性称为全双工。

在进行全双工通信的情况下，在结束通信时如果不小心调用了 close，那么用来读取数据的通信链路和用来写入数据的通信链路就会被同时关闭。只关闭这种全双工的文件描述符的部分通信链路时可以使用 shutdown 系统调用。

为了让 Streem 支持套接字，我对 I/O 处理稍微做了修改，规定进行双向通信时首先调用 shutdown 系统调用。当然，在对不是全双工的文件描述符使用 shutdown 系统调用时会发生错误，不过没有什么危害，所以可以忽略。

服务器端套接字

接下来我们看一下服务器端的套接字的用法。服务器端的套接字通信步骤如下所示。

1. 获取连接方的信息
2. 创建套接字
3. listen/bind
4. accept
5. 输入输出

“获取连接方的信息”的部分与客户端套接字一样，使用的是 getaddrinfo() 函数。不过

如果不考虑同时支持 IPv6 和 IPv4，也可以不使用 getaddrinfo()，直接进行 socket() 和 bind() 系统调用。

要想作为服务器等待客户端连接，就需要用 listen 系统调用指定等待队列的长度，然后用 bind 系统调用注册服务器。

向已注册的服务器端套接字进行 accept 系统调用后，就会返回一个与客户端连接的新的套接字（文件描述符）。需要注意的是，服务器端的套接字与客户端的套接字不同，不能成为输入输出的对象，只用来等待客户端连接。

而与客户端连接的套接字是可以进行普通的读写操作的套接字，所以能够使用 read/write 进行通信。

套接字的使用方法（服务器）

由于服务器端套接字的连接步骤与客户端有些不同，所以提供服务器功能的 Streem 的 tcp_server 函数的实现也与 tcp_client 大不相同。

这么说来，我还没有正式介绍 Streem 的任务创建方法，这里就一并讲解了吧。

tcp_server 函数的实现如图 4-5 所示。这个函数负责生成套接字，然后创建用这个套接字等待连接的任务。

```
struct socket_data {
  int sock;
  strm_state *state;
};

static int
tcp_server(strm_state* state, int argc, strm_value* args, strm_value *ret)
{
  struct addrinfo hints;
  struct addrinfo *result, *rp;
  int sock, s;
  const char *service;
  char buf[12];
  struct socket_data *sd;
  strm_task *task;

  if (argc != 1) {
    return STRM_NG;
  }
  if (strm_int_p(args[0])) {
```

```
    sprintf(buf, "%d", (int)strm_value_int(args[0]));
    service = buf;
  }
  else {
    volatile strm_string* str = strm_value_str(args[0]);
    service = str->ptr;
  }

  memset(&hints, 0, sizeof(struct addrinfo));
  hints.ai_family = AF_UNSPEC;    /* Allow IPv4 or IPv6 */
  hints.ai_socktype = SOCK_STREAM;/* Datagram socket */
  hints.ai_flags = AI_PASSIVE;    /* For wildcard IP address */
  hints.ai_protocol = 0;          /* Any protocol */

  s = getaddrinfo(NULL, service, &hints, &result); ◄— getaddrinfo()函数
  if (s != 0) {
    node_raise(state, gai_strerror(s));
    return STRM_NG;
  }

  for (rp = result; rp != NULL; rp = rp->ai_next) {
    sock = socket(rp->ai_family, rp->ai_socktype, rp->ai_protocol);
    if (sock == -1) continue;

    if (bind(sock, rp->ai_addr, rp->ai_addrlen) == 0) ◄— bind()函数
      break;                    /* Success */
    close(sock);
  }

  if (rp == NULL) {
    node_raise(state, "socket error: bind");
    return STRM_NG;
  }
  freeaddrinfo(result);

  if (listen(sock, 5) < 0) { ◄— listen()函数
    close(sock);
    node_raise(state, "socket error: listen");
    return STRM_NG;
  }
```

```
  /* 创建任务 */
  /* 任务数据的分配与初始化 */
  sd = malloc(sizeof(struct socket_data));
  sd->sock = sock;
  sd->state = state;
  /* 使用strm_task_new创建任务 */
  /* strm_task_new(任务类型，任务函数，结束函数，数据) */
  task = strm_task_new(strm_producer, server_accept, server_close, (void*)sd);
  /* 创建任务对象并赋值给作为返回值的变量 */
  *ret = strm_task_value(task);
  return STRM_OK;
}
```

图 4-5　tcp_server 函数

tcp_server 函数的实现与图 4-4 的 tcp_socket 函数的实现差别不大，差异主要体现在以下几点。

首先，客户端需要指定要连接的主机，而服务器端由于可以接收来自任意主机的连接请求，所以不需要指定主机。其次，在 hints 参数中设置 AI_PASSIVE 选项，这就显式地声明了可以接收来自任何地址的连接。

服务器端套接字用 bind 系统调用代替 connect 系统调用。connect 发出的是“进行连接”的指令，而 bind 则表示“等待连接”。

在 bind 之后调用的是 listen 系统调用。listen 的参数是等待队列的长度，以前有人告诉过我要先把 listen 的参数设为 5，所以这次我把参数设为了 5。不过这是 20 多年前的说法了，也没什么根据，在现在的大流量环境中，设置为更大的数值可能会比较好。

任务的创建

在 tcp_server 的末尾创建任务。首先是分配任务需要的数据并对其进行初始化。

之后，调用 strm_task_new 函数创建新的任务。strm_task_new 函数的参数有 4 个，分别是任务种类、任务函数、结束函数和任务数据。任务种类是表 4-1 的 3 种类型之一。由于这次的任务是“创建”已 accept 的客户端套接字，所以就以生产者的方式来处理，将种类设为 strm_producer。

任务函数是执行实际任务的函数，结束函数是任务结束时被调用的函数。这次指定了 server_accept 和 server_close 函数，我会在后面对它们的具体内容进行说明。

表 4-1　任务种类

种类	含义
strm_producer	生成者（没有输入，有输出）
strm_filter	加工者（有输入，有输出）
strm_consumer	消费者（有输入，没有输出）

从服务器端套接字的 accept 开始的处理均由 server_accept 函数（以及之后被调用的 accept_cb 函数）负责（图 4-6）。客户端连接到服务器端套接字时，服务器端套接字会变为“等待读取”的状态。这时调用 accept() 函数，就可以拿到与客户端连接的套接字。

```
static void
accept_cb(strm_task* task, strm_value data)
{
  struct socket_data *sd = task->data;
  struct sockaddr_in writer_addr;
  socklen_t writer_len;
  int sock;

  writer_len = sizeof(writer_addr);
  sock = accept(sd->sock, (struct sockaddr *)&writer_addr, &writer_len);
  if (sock < 0) {
    close(sock);
    if (sd->state->task)
      strm_task_close(sd->state->task);
    node_raise(sd->state, "socket error: listen");
    return;
  }

#ifdef _WIN32
  sock = _open_osfhandle(sock, 0);
#endif
  strm_io_emit(task, strm_ptr_value(strm_io_new(sock, STRM_IO_READ|STRM_IO_
WRITE|STRM_IO_FLUSH)),
               sd->sock, accept_cb);
}

static void
server_accept(strm_task* task, strm_value data)
{
  struct socket_data *sd = task->data;

  strm_io_start_read(task, sd->sock, accept_cb);
}
```

图 4-6　server_accept 函数

server_accept 函数会取出任务数据，调用 strm_io_start_read 函数，等待对套接

字文件描述符的读取。当客户端套接字的连接请求到来时，调用被指定为回调函数的 accept_cb 函数。

accept_cb 函数调用 accept 系统调用，将得到的套接字封装到 strm_io 结构体，然后进行 emit，这样就把套接字传给了管道中的下一个任务。针对第 3 个参数指定的文件描述符，如果有输入进来，strm_io_emit 函数会调用第 4 个参数指定的回调函数。这里把 accept_cb 本身指定为了回调函数，所以整体上就形成了每当服务器端套接字有输入时就调用 accept_cb 这种循环。

小结

本节我们在 Streem 中增加了套接字通信功能。能够通过网络进行通信的功能扩展了 Streem 的应用范围。

这次套接字的实现也是在 mattn 先生发给我的 Pull Request 的基础上完成的，非常感谢他。

时光机专栏

欢迎加入Streem开发社区

本节是 2015 年 8 月刊中刊登的内容，为 Streem 实现了套接字通信功能。

虽然完成了网络通信的开发，但本节的重点并不是套接字的使用方法，而是 Streem 是如何实现套接字的。因此我觉得本节的内容对学习套接字编程来说帮助不大，尽管我在写稿子的时候觉得讲解部分写得还不错。

不过，今后当有人向 Streem 添加功能时，本节内容可以在 Streem 的 C API 的使用方面为他们提供参考。如果今后有人积极参与 Streem 的开发那就太好了。大家在社区中互相协作一起开发才是开源软件开发的乐趣所在，才是最有意思的事情。

4-2 基本数据结构



数据结构在编程中非常重要。编程语言通过预置几种数据结构来为用户提供编程上的支持。本节我们将看一下各种编程语言的数据结构，然后探讨一下Streem的数据结构该如何设计。



绝大多数编程语言拥有内置的数据结构。这里所说的内置是指预置在语言（处理器）中，不用加载库等就可以使用。

内置什么样的数据结构，取决于语言设计者和语言处理器的开发者，他们的决定会强烈反映出语言的特性。本节我们将看一下各个语言内置的数据结构，以及语言设计者是如何进行设计的。之后再探讨 Streem 的内置数据结构应该如何设计。

■ C 的基本数据结构

首先来看一下 C 的基本数据结构。之所以选择介绍 C，首先是因为 C（和 C++）的基本数据结构与很多语言有很大的不同，非常有特色，其次是因为负责解说的我使用时间最长的语言就是 C。

C 的基本数据结构如表 4-2 所示，大体上可分为 4 组。

表 4-2　C 的基本数据结构

类型	种类	解释	组
char	整数	字符（8 位整数）	整数
short	整数	短整数	整数
int	整数	整数	整数
long	整数	长整数	整数
long long	整数	长长整数	整数
enum	枚举型	实际上是整数	整数
float	浮点数	单精度	浮点数
double	浮点数	双精度	浮点数
*	指针	实际上是地址	地址
[]	数组	实际上是地址	地址
struct	结构体	-	结构体
union	联合体	-	结构体

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

第 1 组是整数。C 的整数涵盖了各种大小。C 没有固定整数的大小，只是像下面这样规定了各种类型的大小关系。

```
char ≤ short ≤ int ≤ long ≤ long long
```

据说以前出现过把所有整数类型的大小都固定为 64 位的特殊的计算机。表 4-3 展示了现在广泛使用的计算机（和操作系统）中常见的组合形式，架构分为 32 位和 64 位，64 位中又包含了两种类型。

整数组中有一个像是“附带品”一样的枚举型 enum。它是表示“名称”的数据类型，但实际上是整数。

表 4-3　C 的整数的大小

架构	32 位	64 位	
char	8	8	8
short	16	16	16
int	32	32	32
long	32	32	64
long long	32	64	64

可以用指针进行运算

第 2 组是浮点数。虽然 C 标准中没有规定，但现在大多数计算机采用了 IEEE754 规格的浮点数格式，单精度是 32 位，双精度是 64 位。

第 3 组是地址。表示内存上某个地点的是指针，表示数据的排列的是数组。通常情况下，数组用来分配内存，指针用来操作内存。如果把数组传递到需要用到指针的地方，数组就会被自动转换为指针。

C 的指针的独特之处在于能够进行和整数相似的运算。指针与整数相加、计算指针之间的差等都是 C 程序员熟悉的功能，但在其他语言中并不多见。

C 的基本数据结构的最后一组是结构体。结构体是用 struct 定义的“数据块”。数组是同一种数据的排列，而结构体则是任意数据的排列。结构体中包含的各个数据（成员）都有名字。虽然在程序上看不出来，但是为了便于内存访问，有时可能在数据和数据之间进行填充（padding）。

自由转换的联合体

在这次的分组中，联合体也包含在了结构体中。联合体用 union 定义。union 的定义与结构体非常相似，但结构体定义的是数据的排列，而联合体定义的是把同一块内存空间解释为不同的类型。

联合体并不是很常用，恐怕很多人不知道它是用来做什么的。联合体有很多用法，典型的有以下 3 种。

- 确保最大的内存空间
- 带条件的结构体定义
- 对内存进行解释的操作

“确保最大的内存空间”是指在有多种数据类型的情况下，无论是哪一种数据类型，都确保有足够的空间保存。

比如在 CRuby 中，对于能够保存各种对象的数组，为了便于内存管理，就使用了表示对象的结构体的联合体数组。实际使用时会根据对象的类型，将数组元素的指针转换（cast，类型转换）为表示对象的结构体的指针。

“带条件的结构体定义”是指结构体的定义根据条件进行变化。这一点需要结合具体的例子来理解。还是以 CRuby 为例。为了减少内存消耗，CRuby 的字符串进行了各种优化。字符串在一定长度以下时在结构体内部保存字符串信息，否则在另外分配的内存空间保存字符串信息。

也就是说，表示字符串的结构体（struct RString）的定义会根据字符串的长度这一条件而发生变化。实现了这一特性的 struct RString 的定义如图 4-7 所示（这里进行了简化）。

```
#define RSTRING_EMBED_LEN_MAX ((int)((sizeof(VALUE)*3)/sizeof(char)-1))
struct RString {
    struct RBasic basic;
    union {
        struct {
            long len;
            char *ptr;
            long capa;
        } heap;
        char ary[RSTRING_EMBED_LEN_MAX + 1];
    } as;
};
```

图 4-7　struct RString

检查 `basic.flags` 的部分就可以得知内部是否保存了字符串信息。在内部保存了字符串信息时访问 `as.ary`，否则访问 `as.heap`，据此实现带条件的定义。

最后一个用法“对内存进行解释的操作”是指不考虑类型信息，直接访问实际内存中的数据。CPU 保存由多个字节组成的整数时的顺序称为“字节序”（endian）。比如，假设构成 32 位整数的 4 个字节从前到后的顺序是 `a`、`b`、`c`、`d`，那么按照 `a`、`b`、`c`、`d` 的顺序保存的方式称为大端（big-endian），而按照 `d`、`c`、`b`、`a` 的顺序保存的方式则称为小端（little-endian）。

正常情况下会考虑使用大端的方式，但实际上采用小端方式的 CPU 占大多数。采用小端方式的 CPU 的代表是 Intel x86，采用大端方式的 CPU 的代表是 SPARC。图 4-8 的程序使用联合体来判断运行中的 CPU 的字节序[①]。

① 这个程序将小端格式之外的格式都判定为大端格式。在很古老的 CPU 中还存在一种按照 `b`、`a`、`d`、`c` 的顺序排列的中端（middle-endian）格式，所以严格来说该程序的判断方法是错误的。不过既然现在连大端格式都要消失了，我觉得也不用在意这个问题。

从数据结构看 C 的特点

上面我们大致了解了 C 的基本数据结构，大家有发现什么吗？

C 的基本数据结构的其中一个特点是表示整数的类型覆盖了从 1 字节（8 位）到 8 字节（64 位）的各种大小，而且同一种大小的类型又包括有符号数和无符号数两种。另一个特点是之前介绍过的允许对指针（地址）进行与整数相同的运算。

包含这些特点在内，C 的基本数据结构直接反映了 CPU 的功能。大多数 CPU 具备进行各种大小的整数的运算的指令，也具备处理浮点数的功能。

对（大多数）CPU 来说，数据是没有类型的，需要根据使用的指令来确定数据（字节或者字节序列）的含义。在 CPU 看来，指针也不过是表示地址的整数而已，既然是整数，那么对其进行整数运算也就没什么稀奇的了。

同样，联合体也不过是用来改变对内存空间的解释而已，看上去复杂的处理，从 CPU 的角度考虑则是再正常不过的事情。

由此我们可以得知，C 是一种在具备一定程度的可移植性和类型检查功能的基础上能够直接操纵 CPU 行为的语言。

在 C 语言出现之前，人们需要用汇编语言为各个硬件单独开发操作系统，而 C 语言的目标是成为一门具有可移植性的高级语言，用 C 语言编写的操作系统的代码基本上不用改动即可运行在不同的硬件上。这一特点也反映在了它的基本数据结构上。

```
#include <stdio.h>
#include <stdint.h>

int
little_endian()
{
  union {
    /* 为了正确地匹配大小
       不使用char、int
       而使用uint32_t（32位），
       uint8_t（8位） */
    uint32_t i;
    uint8_t c[4];
  } u;

  u.i = 0xa0b0c0d0;
  return u.c[0] == 0xd0;
}

int
main()
{
  if (little_endian())
    puts("little endian");
  else
    puts("big endian");
  return 0;
}
```

图 4-8　判断字节序的程序

■ Ruby 的基本数据结构

接下来看一下 Ruby 的基本数据结构。Ruby 内置的数据结构要比 C 多得多，所以我们只看那些具有代表性的。表 4-4 列举了 Ruby 的基本数据结构。

与 C 相比，Ruby 隐藏了 CPU 处理的原始的数据结构，提供了抽象度更高的数据结构。从这一点就可以看出两种语言性质的不同。

Ruby 是作为以文本处理为主的脚本语言诞生的，其数据类型中包含了拥有丰富功能的字符串和正则表达式这一点就充分反映了 Ruby 本来的用途。最近 Ruby 常被认为是 Web 应用的开发语言，

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

用户对正则表达式等文本处理的要求也在逐渐降低。在 Ruby 诞生之后的这二十多年来，它的使用方法在不断发生改变，这让我觉得很有意思。

表 4-4 Ruby 的基本数据结构（部分）

类	种类	解释
`Fixnum`	整数	31 位或 63 位整数
`Bignum`	整数	多倍长度的整数
`Float`	浮点数	只有双精度
`String`	字符串	
`Regexp`	正则表达	/abc$/ 等
`Array`	数组	
`Hash`	散列	也称为关联数组
`Range`	范围	1..2 等
`Proc`	闭包	也称为代码块
`Object`	对象	持有实例变量
`Symbol`	符号	表示标识符的类型

应该把整数型合并为一个

在 Ruby 的基本数据结构中，我认为不太妥当的是 `Fixnum` 与 `Bignum` 的区别。

这两种数据结构都是用来表示整数的。`Fixnum` 是能够表示指针大小的整数，`Bignum` 是用来表示超出 `Fixnum` 范围的整数。在实现上，`Fixnum` 是直接保存在引用（指针）中的整数，优点是不用进行对象分配，从而节省内存。而 `Bignum` 是在堆中分配的对象，虽然多占用了一些内存，但是能够表示的整数范围没有限制（图 4-9）。

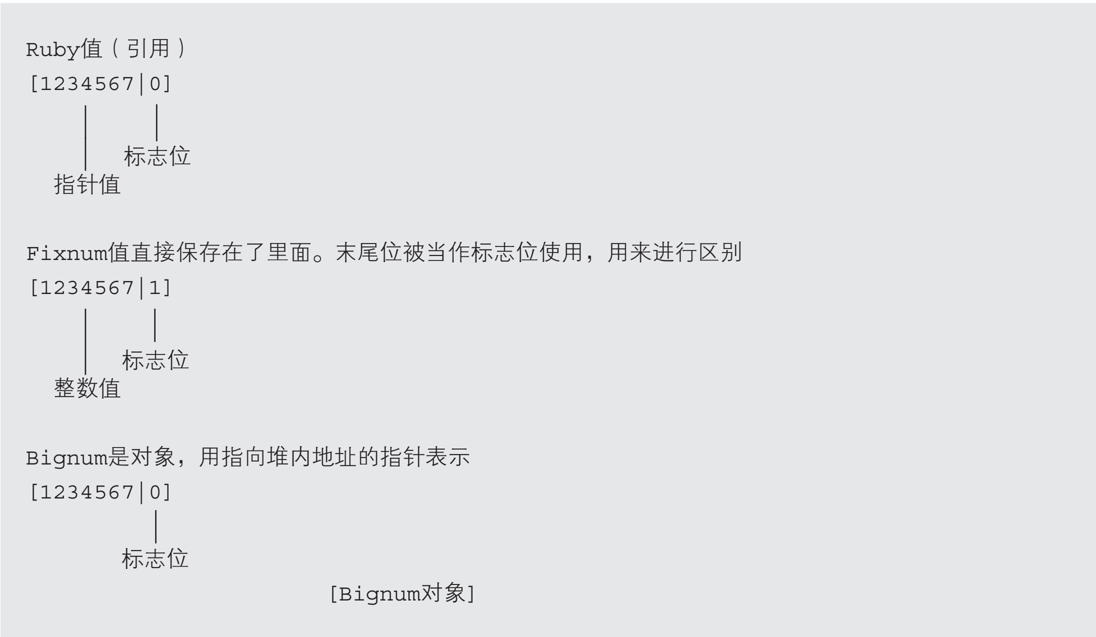


图 4-9 Fixnum 和 Bignum

Ruby 的值是用 C 的指针实现的。大部分操作系统的指针值是 4 或 8 的倍数，所以末尾 2、3 位是 0。末尾位用作标志位，等于 0 时为指针

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

CRuby 就是用带标签的指针实现了 Ruby 的各种数据类型。一般的对象用在堆中分配的结构体实现，值就是结构体的指针。不过在实现了 CRuby 的 C 语言中，作为指针使用的地址可以和整数相互转换。CRuby 就利用了这一特性，把小的整数直接保存在了指针值里。

具体来说，因为 CPU 内存访问的关系，指针值是 4 或 8 的倍数，转换为整数时末尾的 2、3 位一定是 0。利用这一特性，把最后一位当成标志位使用，并用其余位保存整数值。这样一来，比指针的长度少一位的整数（32 位架构的话是 31 位，64 位架构的话则是 63 位）就会作为 `Fixnum` 直接保存在指针值里。

对于超过 `Fixnum` 范围的整数，则在堆中分配 `Bignum` 对象，把整数保存在这个对象中。

严格来说，`Bignum` 可以表示所有长度的整数，但如果运算结果在 `Fixnum` 的范围之内，则会自动转换为 `Fixnum`，这就形成了根据值的范围使用不同的数据类型进行保存的形式。

从含义上来说，不管是 `Fixnum` 还是 `Bignum`，它们表示的都是整数。只是由于实现的不同，在某个范围内的整数用 `Fixnum` 表示，而超过这个范围的整数则用 `Bignum` 表示。

像这种整数用两种类型表示的方式早在 Lisp 中就实现了，所以包括类名在内，Ruby 都是从 Lisp 那里继承的。根据实现的不同对整数进行分类，对语言的开发者来说非常重要，但对语言的使用者来说就没那么重要了。不重要的区别却影响了语法，这就不太好了。

浮点数也是一种数据类型

比如 Ruby 2.0 中进行了优化，在 64 位架构的机器上像 `Fixnum` 一样把一定范围内的 `Float` 值保存在指针值里。因此，即便同是 `Float` 类，也会分为保存在指针值内的值和在堆中分配的对象这两种，但是这个区别是在内部进行处理的，用户不需要关心这个问题。现在想想，`Fixnum` 和 `Bignum` 也应该像 `Float` 一样，只准备一个表示整数的类，在内部进行切换，不让用户看到。

抽象度更高的 Lua 等语言甚至没有对整数和浮点数加以区分，只用 `Number` 这一种类型来表示。考虑到浮点数存在误差的问题，我也犹豫过是否不对二者加以区分，不过这也不失为一个改善的方向。

另外，我也重新审视了从 Lisp 继承来的符号。在 2-6 节提到过，区分使用符号和字符串的做法现在已经过时了。符号和字符串在性能和实现方面都很相似，却使用了不同的数据结构，这并不是现在这个年代提倡的做法。

这种能运行就尽量不进行区分的做法是 Ruby 的一个设计思想，也称为“大类主义”。

OCaml 的基本数据结构

当然也有提倡区分使用的语言，OCaml 就是其中之一。OCaml 是有名的函数式语言，由于其开发效率高，性能较好，所以最近被欧美一些国家的金融行业采用。OCaml 和 Haskell 同为函数式语言，虽然二者在静态类型和强大的类型推导能力方面很相似，但 OCaml 不会自动延迟计算，在

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

需要时也可以执行有副作用的操作，从这些方面来看，可以说 OCaml 更加灵活。

比起 Haskell，我个人更喜欢 OCaml，这并不是说 OCaml 更加出色，只是我个人的喜好而已。

OCaml 与 Ruby 正好相反，提倡区分使用，所以提供了多种类似的数据结构（表 4-5）。

表 4-5　OCaml 的类似的数据结构

类型	含义	解释
`'a list`	链表	线性链表
`'a array`	数组	时间复杂度 $O(1)$，可以修改
`'a * 'b`	元组（tuple）	由多个值组成的数据结构
`{name: 'a}`	记录（record）	相当于结构体

OCaml 的链表如下所示。

```
[ 值 ; 值 ]
```

在链表开头添加值时，使用运算符“::”。

```
0 :: [1; 2]
=> [0; 1; 2]
```

链表是由名为线性链表的数据结构实现的。访问元素的时间与链表的长度成正比，在链表开头添加新元素只需花费很小的开销就能实现（图 4-10）。

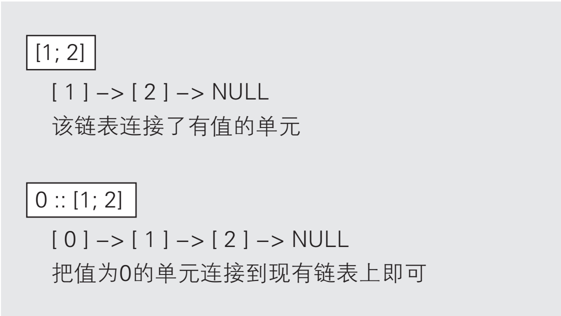


图 4-10　OCaml 的链表

而数组是值的序列。访问数组的第 n 个元素时，时间是固定的，与 n 的大小无关。不过在向数组添加元素时，需要复制整个数组。

与没有显式的类型声明的 Ruby 等语言不同，在静态类型语言 OCaml 中，数组和链表的全部元素在编译时都需要有指定好的共同的类型。这是为了能够在元组（以及记录）中存放多种类型的值。简单来说，把各种类型的值集中在一起的是元组，为每个值赋予名称的是记录。

根据访问开销和类型区分使用数据类型，可以说是 OCaml 独有的风格。虽然与 Ruby 的大类主义不同，但也不失为一种利用静态类型的优势编写高效代码的做法。另外，想到 OCaml 诞生的时期（1996 年，其前身 Caml 是 1985 年），就能明白为什么 OCaml 是这种风格了。

■ Streem 的基本数据结构

看完了其他语言的基本数据结构，下面我们就来思考一下 Streem 的基本数据结构。

首先是数。Streem 并不是作为系统编程语言设计的，所以没有必要像 C 一样直接处理 CPU 操作的各种大小的整数。不考虑区分使用，积极地进行整合反而会比较好。因此，所有数都用 `Number` 类型表示，在内部将整数和浮点数分开，以提升性能。

虽然 C 语言把字符串当作字符型（8 位整数）的数组进行处理，但在 Streem 中，字符串是一种非常重要的数据类型，所以它和 Ruby 一样引入了专用的字符串类型。Streem 也会（在今后阶段性地）实现与 Ruby 相同的字符串操作的方法，但是 Streem 受函数式语言的影响比较大，字符串类型是不可变的，所以不提供修改字符串的功能。前面也说过，不区分使用符号和字符串，在内部把注册到专用表的字符串当作符号使用。

令人烦恼的是数组。Ruby 提供了数组、散列和对象等将多个值集合在一起的数据结构。OCaml 提供了更多的数据结构。当然，这些数据结构都是根据不同的使用场景来区分使用的，但是从其他语言的情况来看，我希望 Streem 可以尽量不要让用户自己去区分使用不同的数据结构。

Streem 不需要链表

将多个值集合在一起的数据结构包括根据访问模式区分的数据结构（通过整数索引访问的数组和把名称作为键的散列）、根据访问开销区分的数据结构（$O(n)$ 的链表和 $O(1)$ 的数组、散列）和根据类型区分的数据结构（单一类型的链表、数组和拥有多个类型的元组）（表 4-6）。

表 4-6　拥有多个值的数据结构

数据结构	访问开销	类型[①]	解释
链表	$O(n)$	单一	在开头添加元素的访问开销为 $O(1)$
数组	$O(1)$	单一	在添加元素时由于需要复制，所以此时的开销为 $O(n)$
散列	$O(1)$	单一	以存在副作用为前提的数据结构
记录	$O(1)$	多个	相当于结构体（Ruby 中是对象）
元组	$O(1)$	多个	内部用数组实现

那么，尽量避免进行区分使用的 Streem 应该采用哪种数据结构呢？

首先不需要采用的就是散列。仔细想想，如果不需要修改数据，散列这种数据结构也就没什么用，所以原则上不会产生副作用的 Streem 不需要使用散列。Streem 通过给元素添加标签的功能来代替散列，带标签的数组也可以代替记录（Ruby 中的对象）。

还有就是数组和链表的区别。虽然数组和链表在拥有多个值这一点上起到的作用相同，但由于内部实现不同，所以访问开销也不同。既然 Streem 的设计方针是避免因实现的不同而要求用户使

① 静态语言的情况。

用不同的数据结构，所以我也就不想保留这个区别了。

可行的设计方案有两个：一个是放弃其中一个，把数据结构统一为一种；另一个是在内部分为两种数据结构，尽量向用户隐藏内部实现。

这里我们看看 Ruby 的做法。Ruby 除了在开发最开始的阶段（公布之前）以外，都只提供了数组，没有提供链表类型，但我还没有听说过 Ruby 因为没有链表而运行开销变得很大之类的事情。虽然 Ruby 过去发生过访问开销过大的问题，但都没有发展到很严重的地步。

之前我觉得将来或许会用到链表，所以就着手去实现了 Streem 的链表。不过经过这次的探讨，我得出的结论是没有必要引入链表，因此还需要整理链表相关的代码，以保持源代码的整洁。

Streem 的其他数据结构

前面讲了数值、字符串和数组的相关内容，除了这些之外，Streem 还有布尔值、闭包、I/O 以及任务等数据结构。今后可能也会根据需要去增加相应的数据结构，但不论怎样，我都想要彻底贯彻之前提到的方针，尽可能不进行区分使用，一种用途只对应一种数据结构（类型）。

本节修改的地方

本节主要对数据结构进行了探讨，并没有怎么修改 Streem 语言处理器的代码。

修改的地方有统一整数和浮点数、整理未开发完成的链表的代码、优化数组内容的字符串显示等。本节对应的源代码的标签是 `201510`。

小结

本节试着从语言内置的基本数据结构的角度对语言的特性进行了解读。C 和 Ruby 的基本数据结构直接反映了各自的设计初衷，这一点很有趣。另外，我们也得出了对语言中的基本数据结构少做区分为好的结论。

在该结论的基础上，我重新审视了 Streem 的基本数据结构，使 Streem 离易于使用的语言更近了一步。今后我也会不断改善 Streem。

时光机专栏

Ruby中也有错误

本节是 2015 年 10 月刊中刊登的内容。为了方便说明，本节在本书中先于 2015 年 9 月刊的文章出现了。

在本节中，我们先总览了各种语言（C、Ruby 和 OCaml）内置的基本数据结构，然后在此基础上设计了 Streem 的基本数据结构。

构成编程语言的元素中最引人注目的就是语法，但其实语言的特性会受到数据类型和库很大的影响。无论语法多么优秀，只要数据类型和库有所欠缺，这门语言就不会流传下来。

另外，本节还提到了各种不同类型的语言（系统编程语言 C、脚本语言 Ruby 和函数式语言 OCaml）的基本数据结构反映了什么样的设计思想，也许能为大家在设计语言时提供参考。

这节还介绍了 Ruby 的 `Fixnum` 和 `Bignum` 的区别。Ruby 2.4（本专栏执笔时还没有发布）在 2016 年 12 月发布，这一版本终于把 `Fixnum` 和 `Bignum` 统一为 `Integer` 类型了。像 Ruby 这种被全世界广泛使用的语言需要考虑兼容性的问题，所以即使过去的设计中有“错误”，也不能轻易修改。不过这次统一为 `Integer` 的过程还算比较顺利。

4-3 对象表示与NaN Boxing

本节将介绍一下改善语言处理器的数据类型的技术，并参考名为V7的JavaScript处理器来实现NaN Boxing技术。

一个偶然的机会，我得知了有名为 V7 的 JavaScript 语言处理器。我们都知道 V8 是 Google Chrome 内置的 JavaScript 语言处理器（并且成为了 node.js 的核心），但是 V7 还是第一次听说。我查了一下，原来 V7 是嵌入式的小型 JavaScript 处理器（它的实现只有一个文件，有 17 000 行左右），据说运行速度很快（它的目标是成为没有 JIT 的处理器中最快的一个）。

我对弄清楚语言处理器的实现很感兴趣，所以花了点时间研究了一下。这个语言处理器的很多实现都很有趣，其中最吸引我的是对象表示的实现。V7 使用了一项被称为 NaN Boxing 的技术，虽然 mruby 也可以通过编译选项实现这一技术，但是 V7 中的实现更加简练。看来我还需要再深入研究一下。

洁净室设计

V7 采用了 GPL2 和商用双许可证。这就表示，当使用 GPL2 许可证不能满足需要时，就需要联系作者，（有偿）获取商用许可证。采用双许可证的目的是，既想开源，又不想被人白白窃取劳动成果，这一点我们可以理解。在 V7 所在的嵌入式领域，为避免陷入许可证方面的纠纷，有相当一部分人（企业）会去购买商用许可证。

但我们这次不是要在 Streem 中实现 V7，而是要把它当成 Streem 的一个参考，所以我不打算去购买商用许可证。可 GPL2 与 Streem 的 MIT 许可证相冲突，这就导致无法直接复制代码。

所以我决定使用（冒牌的）洁净室设计进行开发。洁净室设计是软件反向工程的一种手法，通过隔离解析团队和实现团队，在不侵犯著作权或泄露企业机密的前提下重新进行实现。这次为了避免使用 GPL 保护的代码，我将一边解析 V7 的源代码一边进行讲解，然后根据这些信息来开发 Streem。不过，因为所有的工作都是我一个人完成的，所以不可能实现完全的隔离，这也只不过是冒牌的洁净室设计而已。

引用的表示方法

首先来介绍一下 V7 和 Streem 这样的语言处理器是如何表示对象的。

CPython（用 C 实现的 Python）等语言处理器用指向结构体的指针表示对象的引用。这种单纯使用指针的做法在访问结构体时速度最快，而且还不会浪费内存。但这种做法有一个问题，那就是即使是整数这种频繁使用的值，也需要分配到结构体，这就导致大量对象被分配。

Tagged Pointer 方法可以有效改善这一问题。该方法利用了在将指针转换为整数时，最后的 2~3 位永远为 0 的特点（在很多操作系统中已实现）。在这几位中塞入类型信息，把整数等部分类型的值直接放在指针里（图 4-11）。Emacs Lisp 和 CRuby 等语言处理器就采用了这种方法。

指针按位表示		含义
[...0000 0000]	→	false
[...0000 0100]	→	nil
[...0000 0010]	→	true
[...0000 0110]	→	undef
[...xxxx xxx1]	→	整数
[...xxxx x000]	→	普通的指针（8字节对齐）

图 4-11　Tagged Pointer 方法（以 Ruby 为例）

Tagged Pointer 方法与仅使用指针的方法在内存使用率上相同，通过把整数和布尔值等频繁出现的值塞入指针，进一步提高了整体的内存使用率。虽然从指针中拿出这些值时需要进行一些位运算，但这点开销微不足道。

现在的 Streem 用结构体来表示对象

mruby（默认）、Streem 和 Lua 等语言使用结构体表示对象的引用。目前 Streem 准备了 `strm_value` 这种类型，它实际上是一个结构体，定义如图 4-12 所示。这个结构体可以保存指针、整数和浮点数。顺便说一下，C 语言的 `union` 可以把多种类型保存在一个字段中。

```
typedef struct {
  enum strm_value_type type;
  union {
    long i;
    void *p;
    double f;
  } val;
} strm_value;
```

图 4-12　表示对象引用的 strm_value 的定义

使用结构体方法最大的好处就是实现比较简单，可移植性较强。结构体方法（与单纯使用指针的方法相同）对 CPU 和操作系统没有任何要求，只要是提供了 C 编译器的环境就可以，而且这种方法还避免了单纯使用指针的方法中大量分配整数对象的问题。

这种方法的缺点是内存使用率不高。mruby 中的 `mrb_value` 结构体在 64 位 CPU 中的大小是 16 字节，在 32 位 CPU 中的大小是 12 字节。而如果使用指针来表示对象，在 64 位 CPU 中的大小就是 8 字节，在 32 位 CPU 中就是 4 字节。很明显，使用这种方法会浪费掉一部分内存。除了实现简单和可移植性强之外，该方法没有其他优点。不过在某些实现上，比如目标是在包括嵌入式在内的所有平台上都能移植的 mruby 的实现，可移植性就是一个不可缺少的条件。

表示对象的最后一个方法是 V7 使用的 NaN Boxing。该技术利用浮点数的结构，用 64 位大小表示对象。

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

IEEE 754

C 标准没有对浮点数的表示进行任何规定，不过现在基本上所有的计算机都采用了 IEEE 754 标准来表示浮点数。据我所知有一种叫 VAX 的计算机没有采用 IEEE 754 标准，不过它是很久之前的机器了，估计现在已经没有人使用。另外，我听说有些很老的大型机用的也是自己独有的浮点数格式。

接下来要介绍的 NaN Boxing 是使用（或者说滥用）IEEE 754 格式实现的。首先我们从格式看起。

IEEE 754 定义的浮点数根据精度的不同可以分为多种类型（表 4-7）。各种类型的构造基本相同，只有保存数据的字节大小不同。这里重点解说一下 NaN Boxing 中使用的 `double`。

表 4-7　IEEE 浮点数的种类

精度	类型名	大小	符号部分 / 指数部分 / 尾数部分长度
单精度	`float`	4 字节	1/8/23
双精度	`double`	8 字节	1/11/52
四倍精度	`long double`	16 字节	1/15/112

浮点数由符号部分、指数部分和尾数部分组成（图 4-13）。从最高位开始，按照各部分的长度依次切分，把各部分数据按照无符号整数解释后得到的值作为 `a`、`b`、`c` 时，浮点数的含义如下所示（`double` 的情况）。

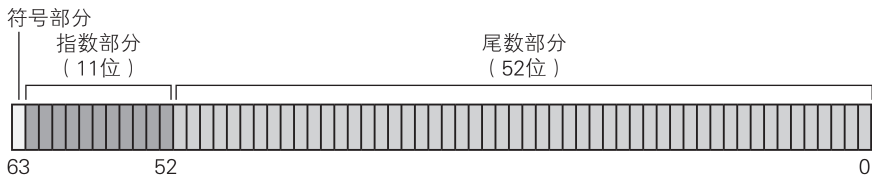


图 4-13　IEEE 浮点数的格式

```
(-1)^a*2^(b+1023)*(1+c/2^51)
```

这个式子的意思是当符号部分为 1 时表示负，为 0 时表示正，指数部分加上偏移量 1023 表示有符号的整数，尾数部分用二进制表示小数点以后的数。

根据这个算式，在用 IEEE 754 的 `double` 表示 2.5 时，由于 2.5 是正数，所以符号部分为 0。按照下面的式子把 2.5 的整数部分转换为 1，这样尾数就是 1.25，指数是 1。

```
1.25 * 2^1
```

尾数部分去掉小数点左边的 1 之后变为 0.25，用二进制进行字节表示，如下所示。

```
01000000（后面 44 位都是 0）
```

指数部分加上偏移量 1023 后值为 1024，用字节表示为

```
10000000000
```

如果整个数从最高位开始按照符号部分、指数部分和尾数部分的顺序排列，那么用十六进制表示时就是

```
0x4004000000000000
```

打印这个值的程序如图 4-14 所示。前面提到过 `union` 允许按照不同的类型解释位模式，这一特性非常适合用在这样的程序里。

```
#include <stdio.h>
#include <stdint.h>

int
main()
{
  union {
    double f;
    uint64_t i;
  } u;
  u.f = 2.5;
  printf("0x%lx\n", u.i);
  return 0;
}
```

图 4-14　把 IEEE 754 的 double 按照十六进制打印的程序

特殊的浮点数

IEEE 754 标准中有两个特殊的值[①]：一个是无穷大（∞），另一个是非法值（Not a Number，NaN）。无穷大在数学上表示无穷大的值，比如非 0 浮点数除以 0 的结果等，有正负两个值。在 IEEE 754 标准中，无穷大的指数部分是 2047，尾数部分是 0。正负则由符号位决定。

`NaN` 用于表示 0/0、∞ +（−∞）等数学上没有意义的计算结果，以及负数的平方根等超出实数范围的值。在 IEEE 754 标准中，`NaN` 的指数部分与∞相同，都是 2047（用位表示为 11111111111），尾数部分为 0 以外的数。

① 其实还有一个名为“非规格化浮点数”（denormal number）的特殊值（组），不过本次不涉及。

NaN Boxing

利用 NaN 的特性，用浮点数来表示对象的方式就是 NaN Boxing。

前面说过，NaN 的指数部分全部为 1，尾数部分为 0 以外的数，这样就会有 2^52 – 1 种，也就是 450 359 962 730 495 种位模式。如果有这么大的存储空间，就可以存储多种类型的值。由于是向 NaN 的空隙塞入值，所以把这项技术称为 NaN Boxing。

但令人意外的是，V7 中表示对象的类型并不是浮点数，而是 64 位整数（uint64_t）。这个用 64 位整数表示的对象类型被 typedef 为“v7_val_t”这一名称。

不使用浮点数而使用整数的原因可能是使用整数更快。我没有做基准测试，所以这只是我的推测而已。在将值作为函数调用的参数或返回值进行传递时，有的 CPU 会对浮点数进行特殊处理，在这样的环境中，使用整数的效率可能会更高。使用 NaN Boxing 技术时只要提供位模式即可，不需要用浮点数本身来表示对象。

V7 把 64 位分成符号部分（1 位）、指数部分（11 位）和尾数部分（52 位），并按照以下规则来解释。

首先是符号部分。包括 V7 在内的很多 NaN Boxing 实现中不使用符号部分，V7 中符号部分永远为 1。根据 NaN 的规定，指数部分也全部是 1，实际的值保存在剩余的 52 位中。

在尾数部分的 52 位中，前面 4 位被作为表示对象类型的标志位使用（表 4-8），剩余的 48 位（6 字节）被用来表示各种对象。

表 4-8　V7 的值的种类

种类	标志位	解释
OBJECT	0x1111	对象
FOREIGN	0x1110	外部指针
UNDEFINED	0x1101	JavaScript 的 undefined
BOOLEAN	0x1100	布尔值
NAN	0x1011	NaN（浮点数）
STRING_I	0x1010	内联字符串（长度 < 5）
STRING_5	0x1001	内联字符串（长度 = 5）
STRING_O	0x1000	字符串（GC 的对象）
STRING_F	0x0111	外部字符串
STRING_C	0x0110	字符串大型对象块
FUNCTION	0x0101	JavaScript 函数
CFUNCTION	0x0100	C 函数
GETSETTER	0x0011	getter + setter
REGEXP	0x0010	正则表达式
NOVALUE	0x0001	数组用的未初始化的值
INFINITY	0x0000	无穷大（浮点数）

总的来说，在 v7_val_t 的 64 位中，符号部分 + 指数部分 + 尾数部分的前 4 位这 16 位是表示对象类型的标签，剩余的 48 位则用来保存其他想要表示的值。

布尔值的保存方法

那么具体如何保存值呢？

我们从最简单的布尔值开始看起。V7 中创建布尔值（true 或者 false）的函数是 v7_create_boolean()，其定义如图 4-15 所示。

```
v7_val_t v7_create_boolean(int v) {
  return (!!v) | V7_TAG_BOOLEAN;
} ▲ !!v在v非0时为1，v等于0时为0
```

图 4-15　v7_create_boolean()

V7_TAG_BOOLEAN 是表示 BOOLEAN（布尔值）的标签（符号部分 + 指数部分 + 标志位）。前面提到过保存值的部分有 48 位，这里就用来保存 true 时为 1、false 时为 0 的布尔值。

假如要表示 Ruby 的符号这种 JavaScript 中没有的值，也需要使用同样的方法。也就是说，把符号对应的整数（48 位以内）与它的标签组合起来，以此表示对象。

整数的保存方法

虽然 JavaScript 中没有整数（所有的数都用浮点数表示），但这里还是大致讲解一下使用 NaN Boxing 保存整数的方法。既然表示类型的符号部分和指数部分合起来有 16 位，用来保存值的有 48 位，那么只要这 48 位能把整数保存下来即可。

最简单的方法就是采用 32 位整数。虽然 48 位中有 16 位就这么被浪费掉了，但是在 32 位 CPU 的时代，几乎所有的计算都是用 32 位进行的，所以应该没有什么坏处。而且比起 48 位这样不大不小的位数，很多 CPU 处理 32 位整数的效率会更高。这也是该方法的一个优点。

另外一个办法是把这 48 位全部用来表示整数，但是在这种情况下就需要用 64 位整数进行计算，需要小心数据溢出，而且处理本身也可能会变得复杂。

浮点数的情况

NaN Boxing 把值保存在浮点数不用的位中，所以不需要对浮点数进行加工。不过，由于表示浮点数的类型是 double，表示对象的是 64 位无符号整数，所以需要进行转换。做法与图 4-14 相同，将浮点数保存到 union 之后，再作为整数取出。

指针的保存方法

在要保存的各种类型的值中，最可能出现问题的是指针。如果是 32 位架构的系统，那么指针

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

的大小也是 32 位，保存在 48 位的数中没有任何问题。但如果是 64 位架构的系统，那么指针的大小就是 64 位，这样就超过了 48 位的范围。

不过幸运的是，大多数操作系统没有把 64 位全部用来表示指针，而是只使用了大约 48 位可以覆盖的值。想想看，有 48 位就可以访问 256 TB 的内存空间，所以目前不会有什么问题。

遗憾的是，部分操作系统（Solaris 等）把 NaN Boxing 用来表示标签的前 16 位空间也用来表示指针了，所以不能在这些操作系统上使用该技术。虽然与过去不同，现在使用 Solaris 的人越来越少，所以也许不会存在太大的隐患，但是 NaN Boxing 最大的短板还是可移植性。

如果指针也可以放进 48 位的范围之内，那么剩下的操作就和布尔值等一样，只需与标签组合起来放进 NaN 中即可。

字符串的保存方法

V7 在字符串的表示上下了不少功夫。由于字符串是频繁使用的对象，所以 V7 细致地考虑了性能问题。表 4-8 列举的 16 种值之中就有 5 种是字符串类型，由此可见一斑。

首先是 STRING_I 和 STRING_5。由于有 48 位的空间可以用于表示值，所以长度在 6 字节范围内的字符串可以直接保存在 NaN 值中。JavaScript 不能修改字符串，因此可以使用这种方法。Ruby 的字符串是可以修改的，所以 Ruby 不能直接使用这种方法。而在 Streem 中，包括字符串在内的对象也是不可修改的，所以 Streem 也能用这种方法。

为了保持与 C 语言字符串的兼容性，必须在 V7 的字符串末尾带上 NUL（'\0'）。这样一来，可以保存的最大字节数就是 5。

1~4 字节的字符串用 STRING_I 表示（图 4-16a）。在 5 字节的字符串的情况下，由于没有空间保存字节数，所以用专门的标签来表示（图 4-16b）。

在剩余的 3 种字符串类型中，STRING_F 是由外部赋与的，不受 V7 的 GC 管理。由于不受 GC 管理，所以会被直接作为指针处理。

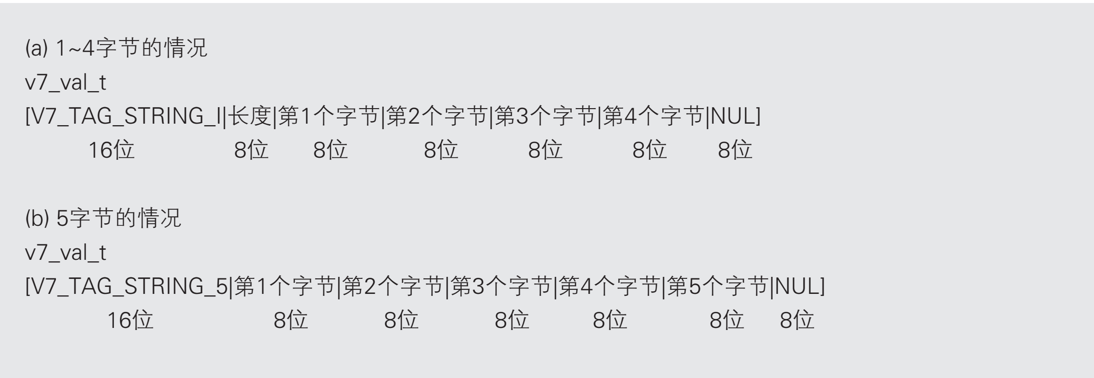


图 4-16　内联字符串

字符串的 GC

STRING_O 的 O 是 owned 的首字母，也就是说，它是由 V7 进行内存空间管理的字符串。字符串在 V7 管理的内存空间中被分配。内存空间不足时 GC 开始启动，字符串占用的内存空间会被回收。

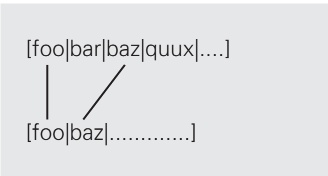


图 4-17　移动压缩

假定只有“foo”和“baz”存活

V7 对字符串采取了移动压缩的方法。具体来说，字符串被回收之后产生了内存缝隙，通过移动字符串后面的值填满缝隙，从而有效地利用内存空间（图 4-17）。但是注意要将对移动前的字符串的引用（地址）修改为移动后的地址，否则数据就会遭到毁坏。

因此，V7 使用了 STRING_C。这种类型比较复杂，乍一看不知道在进行什么处理，我们通过图 4-18 来看一下大致步骤。

(1) v1、v2、v3是对字符串的引用

v1[STRING_O|addr]　v2[STRING_O|addr]　v3[STRING_O|addr]

"foobarbaz"

(2) 扫描v1的引用，将v1的地址写入字符串前面的6字节

v1[STRING_C|foobar]　v2[STRING_O|addr]　v3[STRING_O|addr]

"(v1 addr)baz"

(3) 扫描v2的引用，建立从v2到v1的链接

v1[STRING_C|foobar]　v2[FOREIGN|(v1 addr)]　v3[STRING_O|addr]

"(v2 addr)baz"

(4) 扫描v3的引用，建立从v3到v2的链接

v1[STRING_C|foobar]　v2[FOREIGN|(v1 addr)]　v3[STRING_O|(v2 addr)]

"(v3 addr)baz"

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

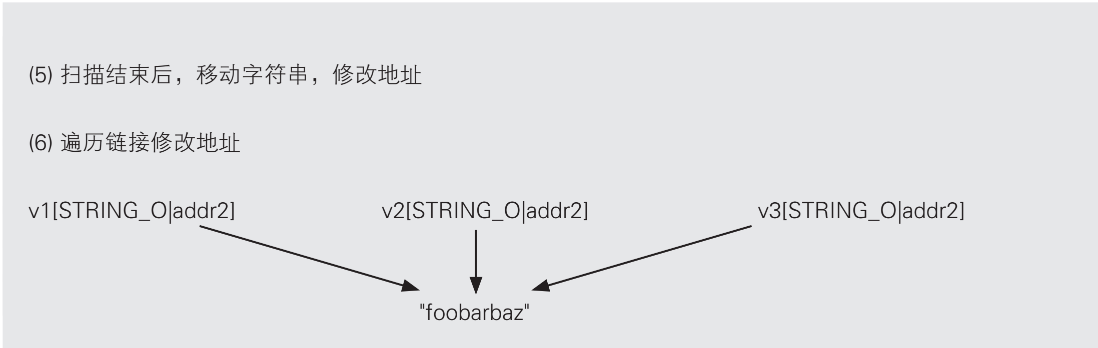


图 4-18 字符串 GC 的步骤

首先，在 GC 检查“存活的”字符串的标记阶段，如果找到了还没有被标记的字符串的引用 `v`，就进行下列处理。

1. 标记字符串
2. 将字符串前面的 6 字节写入 `v`
3. 将 `v` 的标签改为 `STRING_C`
4. 把 `v` 的地址写入字符串

发现已标记的字符串的引用 `v2` 时，进行下列处理。

1. 将保存在已标记的字符串中的地址写入 `v2`
2. 将 `v2` 的标签改为 `FOREIGN`
3. 将 `v2` 的地址写入字符串

重复上述处理，在标记阶段结束时，存活的字符串就会变成以下状态。

- 已被标记
- 对象所有的引用都在一个链表中连在一起
- 链表的末尾是用地址替换的部分的字符串信息

我在读 V7 的代码时，发现这个功能的代码在非小端字节序的 CPU 上可能会出错。不过最近 x86 和 ARM 基本上都是小端格式的，所以我们不需要在意这个问题。

之后就是重复以下过程：依次扫描字符串的内存空间，如果字符串已被标记，就移动字符串填满内存缝隙，遍历链接修改所有引用了字符串的地址。

这个处理看上去非常麻烦，但这么做是有原因的。虽然为了让实现更简单也可以不采用移动压缩等方法，直接 `malloc`（分配）内存空间，使用完之后再进行 `free`（释放），但是一般来说，`malloc` 大量分配比较小的内存会造成内存空间的浪费。另外，由于字符串的地址比较分散，所以

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

工作集（访问的地址的范围）会变大，缓存也不容易发挥作用。考虑到运行效率，想必 V7 的开发者也认为实现这个复杂的处理是值得的。

在 Streem 中引入 NaN Boxing

接下来我们就参考前面介绍的 V7 的 NaN Boxing 的实现，尝试在 Streem 中引入 NaN Boxing。

首先将表示对象的 strm_value 的内部实现由结构体替换为 uint64_t。另外，为了表示对象的类型，我还准备了表 4-9 中的标签。现在只有 11 种（最多 15 种），以后可能会根据需要再进行添加。

字符串使用了与 V7 相同的技术，将 6 字节以内的字符串保存在 strm_value 里。Streem 本来就不需要在末尾放置 NUL，所以能够最大程度地使用 48 位（6 字节）。

不过目前我不打算引入移动压缩的方法。在进行过基准测试之后，如果发现这里确实是影响性能的一个因素，届时再解决也不迟。

表 4-9　Streem 的值的类型

种类	解说
BOOL	布尔值
INT	整数
ARRAY	数组
STRUCT	带有成员名的数组
OBJECT	new 创建的数组
FOREIGN	外部指针
STRING_I	字符串（1~5 字节）
STRING_6	字符串（6 字节）
STRING_O	字符串（GC 管理）
STRING_F	字符串（非 GC 管理）
CFUNC	C 函数

GC 的实现

到目前为止，Streem 使用了面向 C/C++ 的 Boehm GC 库，没有实现自己的 GC 功能。

引入 NaN Boxing 之后，就不能直接看到指针的值了，Boehm GC 也就无法工作了，看来需要引入 Streem 自己的 GC。这次我准备了标记清除法这一非常简单的 GC 算法，今后会再去设计开发能够发挥 Streem 语言特性的 GC。

小结

本节通过分析以实现高性能为目标的 JavaScript 语言处理器 V7 的源代码，介绍了表示对象的 NaN Boxing 技术，我还把这项技术引入到了 Streem 中。读者自己在设计语言或开发语言处理器时，也可以参考这些知识。如果读者之中有人创建了继 Ruby 之后在全世界流行的语言，那我将欣慰至极。

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

时光机专栏

对GC的实现干劲不足

本节是 2016 年 1 月刊中刊登的内容，我们从小型 JavaScript 语言处理器 V7 的代码中了解了 NaN Boxing 的实现方法。在浮点数中保存指针等值的 NaN Boxing 是很高超的技术，所以即便是作为 NaN Boxing 实现的相关资料，本节内容也是很有意义的。

这里需要向大家说明一件事。正文里虽然说“我准备了标记清除法这一非常简单的 GC 算法”，但实际上还没有准备好。当然并不是我有意撒谎，我在写稿子的时候是想去完成它的，但是因为时间（和干劲）的关系没能完成。

以前在开发 mruby 的 GC 时，我利用无聊的会议时间几个小时就开发好了，所以我想这次也肯定能够完成，无奈干劲不足，结果到现在写这个专栏时还没有实现 Streem 自己的 GC。真是不好意思。

4-4 垃圾回收

在4-3节，我采用NaN Boxing技术改善了Streem对象的表示方法。随着这个技术的使用，内存管理的方法也需要相应地进行修改。借此机会我想介绍一下内存管理，特别是GC（垃圾回收）算法的相关内容，并研究一下Streem要如何实现GC功能。

在 Java 和 Ruby 这样的语言中，程序运行期间会创建大量的对象。从计算机的角度来看，对象只不过是储存数据的内存空间而已，而编程语言则把它们看作对象。

同为面向对象语言的 C++ 需要开发者手动管理对象占用的内存空间。像 C 这种面向对象之前的语言也需要手动内存管理，在这一点上它们是一样的。

C 使用 `malloc()` 函数直接分配内存空间，C++ 使用 `new` 在堆空间中分配对象。这些调用过程都要求操作系统预先分配一块内存空间，然后系统从这块内存中分割出本次调用所需要的内存并返回。之所以这么做，是因为如果每次都要请求操作系统分配内存，那效率就太低了。

在如此分配的内存使用完毕后，程序就会用 `free`（C）或 `delete`（C++）来告诉系统这些内存已经不需要了。当不用的内存达到一定值时，系统就会将这些不用的内存还给操作系统。但是，这个“已经不再使用”的状态却是产生各种问题的原因。

自动释放内存空间

如果不小心把还在使用的内存空间还回去了，那么还回去的内存空间在不久之后就会被挪为他用。之后再去访问，这个内存空间中的数据就可能被修改掉了，这就导致程序不能正常工作，甚至异常退出。

反之，如果想着可能还会使用而迟迟不向系统归还内存空间，或者使用后忘记归还，这种情况也会导致问题发生。保留实际上已经不再使用的空间会白白浪费内存，甚至会引发性能下降和异常退出等问题。管理大量分配的零散的内存，本来就不是人擅长的事情。

自动进行内存管理，特别是内存释放的技术称为 GC。GC 其实在很早以前就出现了。在 20 世纪 60 年代就有人研究这项技术，并写了很多论文。虽然该技术在大学研究室中使用已久，但真正在普通开发者中间普及则是在 20 世纪 90 年代 Java 出现以后。在这之前，很少有人知道这项技术。

GC 技术曾经饱受质疑。随着 Java 的出现，人们终于开始关注 GC，但当初很多人对这项技术持否定意见。“GC 不可靠”“分配的内存空间应该由人来显式地释放”“GC 比手动管理内存还要慢，不能使用”等意见不绝于耳。

随着时间一天天过去，Java 得到了普及，这种声音也慢慢消失了。这是因为 GC 技术变得更加先进，证实了 GC 比人直接管理内存出现的错误更少，在多数情况下性能更好。

当然，目前 GC 还不适用于嵌入式实时系统这样的环境，但除了这些特殊情况，GC 已经普遍应用在各种环境中了。

追踪法和引用计数法

GC 中有追踪法（trace）和引用计数法（reference counting）两大方法，这两种方法分别代表了两个极端。

追踪法是指从被称为根的起点开始递归地遍历（追踪）被引用的对象的方法。追踪法又包括一边追踪一边标记存活的对象，最后把没有被标记的（垃圾）对象统一回收的“标记清除法”，以及把在追踪过程中发现的存活的对象复制到别的内存空间，然后清除留在旧内存空间中的对象的“复制法”等。

追踪法的优点是能够检查出从根节点开始被间接引用的存活的对象。但反过来，对象越多，GC 所需要的时间就越长，这也是它的缺点。

标记清除法

标记清除法是最早被开发出来的算法。它的原理非常简单，从根开始递归地标记被引用的对象，然后把没有标记的对象作为垃圾回收。

标记清除法的概要如图 4-19 所示。

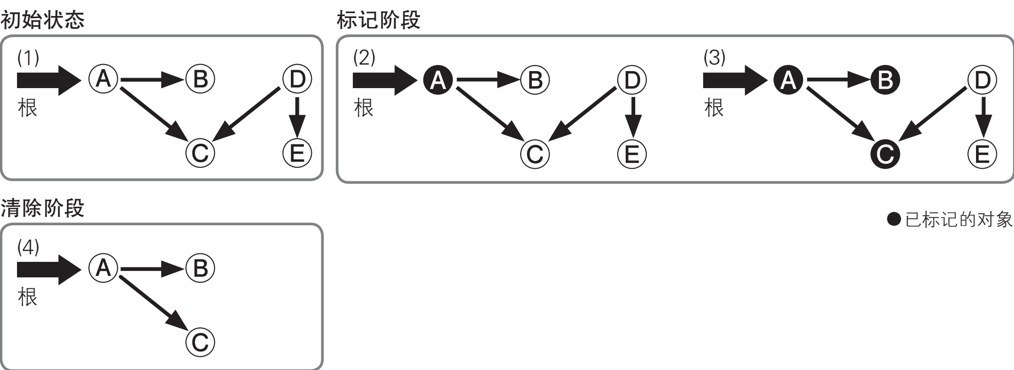


图 4-19　标记清除法

首先，随着程序的运行，对象被分配（图 4-19(1)）。有的对象会引用其他对象。

GC 开始后，从根开始对被引用的对象进行标记（图 4-19(2)）。一般情况下，标记多作为对象内部的标志位实现。这里我们把已标记的对象涂黑。

同样也要对被标记的对象所引用的对象进行标记（图 4-19(3)）。重复这个过程，对能够从根开始间接引用的所有对象进行标记，这一阶段称为“标记阶段”。标记阶段结束时，被标记的对象会被当作存活的对象。

依次扫描所有的对象，回收没有被标记的对象（图 4-19(4)），这一阶段称为“清除阶段”。为了方便下次回收，在扫描的同时也会清除存活的对象的标记。

还有一种名为“标记压缩”（mark-compact）的方法，它是标记清除法的延伸，该方法不会清除没有标记的对象，而是将存活的对象压缩到一起。

在标记清除法及其延伸方法中，处理时间与“存活的对象数”和“所有的对象数”之和成正比。当有大量对象被分配，其中只有一小部分对象存活时，在清除阶段就需要扫描大量已死的对象，这就会浪费一些不必要的时间。这也是该方法的一个缺点。

复制法

复制法是以克服上述缺点为目标的算法。

复制法把从根开始被引用的对象复制到其他内存空间，然后再递归地复制被复制的对象所引用的对象。

图 4-20(1) 所示为 GC 开始前内存的状态，与图 4-19(1) 相同。

接下来，在旧的对象所在的内存空间（称为旧空间）之外再准备一个新的内存空间（称为新空间），然后把从根开始被引用的对象复制到新空间（图 4-20(2)）。

将被复制的对象所引用的对象也顺藤摸瓜式地复制到新空间（图 4-20(3)）。复制完成后，存活的对象全部移动到新空间，已死的对象则留在旧空间。

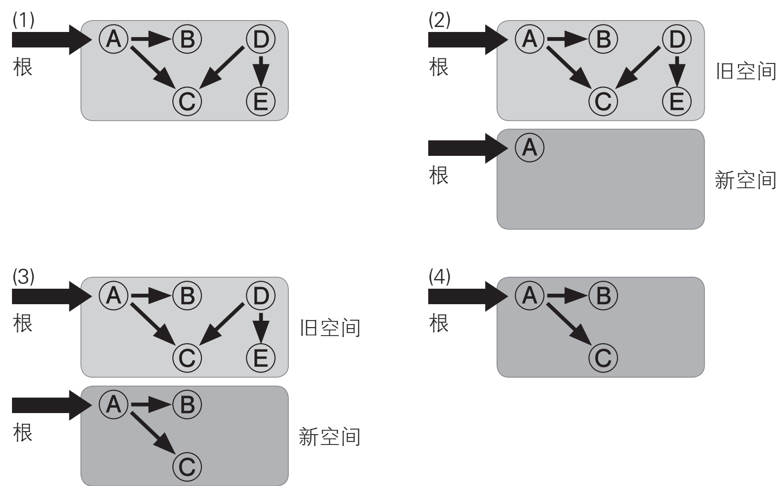


图 4-20　复制法

这时如果释放旧空间，已死的对象所占据的内存空间就会一下子被释放（图 4-20(4)），不需要扫描单个的对象。下一次进行 GC 时，这个新空间就会成为旧空间。

从图 4-20 可以看出，复制法中不存在标记清除法中的清除阶段。在分配了大量对象且其中的大部分对象会马上死去的情况下，标记清除法的清除阶段会产生相当大的开销，而复制法则没有这个开销。但是，比起标记对象，复制对象的开销会更大，所以复制法不适合在存活的对象比重较大的场景下使用。

该算法的另一个优点是“局部性”。复制法是按顺序把被引用的对象复制到新空间的，所以关系相近的对象很有可能被配置到内存空间中临近的区域，这被称为局部性。在局部性较强的情况下，内存缓存更容易起作用，程序的运行效率也更高。

复制法的缺点是内存使用率不高。复制的过程虽然是暂时的，但需要准备两块同样大小的新旧内存空间，这样就只能有效使用最大内存消费量的一半。复制法的延伸方法可以更加细致地分割内存空间，减少内存空间的浪费。

GC 的性能指标

追踪法的基本算法大致上可以分为上面的标记清除法、复制法以及它们的延伸方法。

GC 与程序处理的本质无关，所以花费在 GC 上的时间越短越好，但是上述基本算法在性能上还存在一些问题。

GC 的性能可以用两个指标来衡量：GC 运行时间和停止时间。

GC 运行时间是 GC 处理本身的性能，是指在整个应用程序的运行时间之中，GC 所消耗的时间。

停止时间是指当应用程序中断本来要做的处理转而进行 GC 时，处理被中断的时间。特别是最大停止时间，是一项重要的指标。

停止时间之所以重要，是因为应用程序长时间不响应会出现问题。假如有 1000 个人访问 Web 服务器，Web 服务器在 10 毫秒内对其中的 999 人返回了结果，而剩下的一个人不巧赶上了 GC，花了 10 分钟才等到结果返回。这种情况并不是我们想看到的。再比如控制机器人的软件，如果软件在机器人行走的时候进行了 GC，导致有 1 秒钟的时间失去了对机器人的控制，那机器人可能就会摔倒吧。

辅助的 GC 技巧

将 GC 的基本算法与一些技巧组合使用，可以改善 GC 的性能。这次我向大家介绍两个具有代表性的技巧：分代 GC 和增量 GC。我们也可以组合使用多种技巧。

首先来介绍 GC 技巧中最重要的分代 GC。

分代 GC

分代 GC 是减少程序运行时间中 GC 所消耗的时间的技巧。

分代 GC 的基本思想利用了普通程序的一个特点，即大部分对象会在较短时间内成为垃圾，在一定时间内存活的对象会拥有更长的寿命。如果寿命长的对象容易存活，寿命短的对象早早就不需要了，那么就可以重点扫描刚分配没多久的“年轻”对象。这样一来，即使不扫描全体对象，也能回收很多垃圾。

分代 GC 将对象分为刚分配没多久的“新生代”和长时间存活的“老年代”，根据实现的情况可能还会进一步细分。

只扫描很可能马上就死亡的新生代对象的 GC 被称为 Minor GC。Minor GC 的具体回收步骤如下所示。

首先从根处开始扫描，寻找存活的对象。这个阶段的算法用标记清除法或复制法都可以，但大多数分代 GC 采用了复制法。需要注意的是，扫描期间如果发现了属于老年代空间的对象，就不再扫描之后的对象了，这样可以大量减少要扫描的对象的数量。

然后把还存活的对象分到老年代，具体做法是：在使用复制法的情况下，把对象复制到老年代空间；在使用标记清除法的情况下，一般是给对象加上某种标志位。

记录老年代对新生代对象的引用

这时会出现问题的是老年代空间的对象对新生代空间的对象的引用。如果只扫描新生代空间，那么老年代空间的对象对新生代空间的对象的引用就不会被检查到。因此，只被老年代空间的对象引用的新生代空间的对象就会被当成已死去的对象，所以分代 GC 需要监视对象的更新。如果有老年代空间的对象对新生代空间的对象进行引用，就把这个引用添加到被称为记忆集合（remembered set）的表中。在进行 Minor GC 时，把这个记忆集合也包含进根里。

要想让分代 GC 正常工作，就需要时刻把记忆集合的内容更新为最新状态。因此，当老年代空间的对象对新生代空间的对象进行引用时，就要把记录这个引用的过程加在所有更新对象的地方。这个记录引用的过程被称为写屏障（write barrier）。

老年代空间里的对象一般寿命很长，但这并不代表这些对象是“不死的”。随着程序的运行，老年代空间中死去的对象会越来越多。要想避免出现老年代空间中死去的对象占用内存的情况，就需要不时地对所有对象进行扫描，这种扫描所有空间的对象的 GC 被称为 Full GC 或 Major GC。

分代 GC 减少了 GC 时要扫描的对象的数量，可以缩短 GC 运行时间，但由于 Major GC 的存在，最大停止时间还是没能得到改善。

增量 GC

通过前面机器人的例子我们可以知道，比起 GC 的性能，实时性强的程序更重视最大停止时间的长短。

实时性强的程序需要预测 GC 所产生的中断时间，比如规定 GC 最多不得超过 10 毫秒。

普通的 GC 算法是无法做到这一点的。这是因为 GC 的停止时间取决于对象的数量和状态。因此，为了保持实时性，不再等待 GC 全部完成，而是把处理细分并一点一点地执行，这种做法被称为增量 GC。

在增量 GC 的情况下，由于 GC 处理是一点一点地进行的，所以在 GC 处理的过程中，在程序保持运行的状态下引用被替换。扫描结束后，在已标记的对象被替换，变为引用新的对象的情况下，由于这些新的对象没有被标记，所以即使它们还活着，也会被回收。

为了避免这个问题出现，增量 GC 和分代 GC 一样使用了写屏障。在对已标记的对象进行替换时，利用写屏障把新引用的对象加入到扫描的起点。

由于增量 GC 是把处理细分后执行，所以中断时间可以控制在固定值之内。而中断处理需要花费一定的开销，所以花在 GC 上的总时间会增加。我们需要对二者折中权衡。

引用计数法

与追踪法并列的另一大 GC 方法是引用计数法。引用计数法是指在各个对象中记录该对象的被引用数的方法，每当引用发生变动时就更新这个计数（图 4-21）。

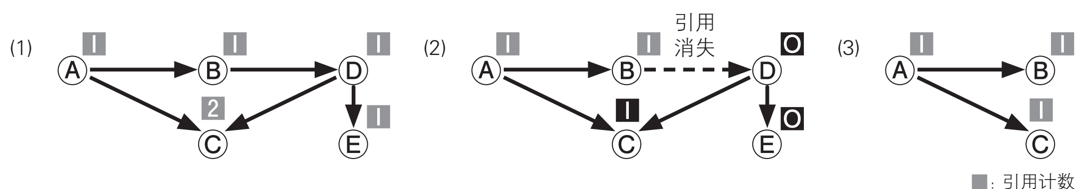


图 4-21　引用计数法

被引用数发生变动的场合有：对变量的赋值、对象内容的更新、函数的结束（局部变量使用的引用消失）等。很明显，对象的被引用数为 0 就意味着该对象没有被任何对象引用，所以就要释放该对象的内存空间。

引用计数法最大的优点是可以从局部范围来判断对象的释放。追踪法需要从根开始扫描全部的对象才能判断对象的生死，而引用计数法在引用为 0 的瞬间就可以判断出该对象已死。另外，引用计数法能够以一个个对象为单位来释放对象，与其他算法相比，GC 产生的停止时间更短（的情况较多），这也是它的一个优点。

当然它也有缺点。引用计数法最大的缺点就是不能释放循环引用的对象。两个相互引用的对象

即使（作为一个整体）没有被其他任何地方引用而成为了垃圾，被引用数也不会变为 0，结果是内存永远都不会被释放。

引用计数法的另一个缺点是，当引用发生变动时，需要正确地增减引用计数，如果不小心忘记了，就会引起很难找出原因的内存问题。

最后，引用计数法还有一个缺点，那就是引用计数的管理与并行处理不能很好地协作。同时在多个线程增减引用计数时，可能会出现引用计数值不一致的情况（结果就会出现内存问题）。为了避免出现这个问题，需要排他地进行引用计数的操作。对于需要频繁进行的引用操作，每次加锁产生的开销也是不能忽视的。

Streem 的 GC

一直以来 Streem 使用的都是非常方便的 `Boehm GC` 库，这个库可以向 C 和 C++ 的程序自动添加 GC 功能。使用这个库之后，对于那些用 `malloc` 和 `new` 分配的对象（内存空间），开发者无须逐个进行 `free/delete`，GC 库就会自动检测出不再使用的对象并回收。但遗憾的是，这个库有使用上的限制，不能对指针进行加工，这样就不能和 4-3 节引入的 NaN Boxing 同时使用了。

所以我只好放弃 `Boehm GC`，去开发自己的 GC。既然要开发自己的 GC，自然就想在实现上尽可能地展现出 Streem 本身的特点。

Streem 的一个特点是，在事件循环开始后，各个处理以任务为单位独立运行。任务运行中产生的数据，除了显式调用了 `emit` 的对象（和该对象递归引用的对象）之外，都不会被其他任务访问。这样一来，只要标记由 `emit` 向其他任务“输出”的对象，在任务结束时将其余创建的数据全部删除，就有可能实现以任务为单位的 GC。也就是说，这种 GC 和分代 GC 一样，不用扫描全体对象就可以进行，因此 GC 运行时间就会缩短。

旧的对象不能引用新的对象

Streem 的另一个特点是几乎所有的数据结构都不能修改，数据结构不能修改对 GC 来说有以下几点好处。

首先，对象不能修改就意味着该对象所引用的对象必须在对象创建时就已经存在。也就是说，旧的对象无法引用在它后面创建的新的对象，于是也就不会产生循环引用的问题了。前面提到过，引用计数法的缺点是无法处理循环引用的情况，而不可修改的对象则可以克服这个缺点。

另外，既然旧的对象无法引用比它新的对象，那么在实现分代 GC 时也就不需要使用写屏障了。

这一特点对实现复制法也有好处。如果对象可以修改，就需要严格区分对象和它的副本。这是因为即使修改了某一个对象，它的副本也不会被修改。而如果禁止修改对象，（除非是显式地用指针值等进行区分）就没有必要区分对象的原本和它的副本了。一般来说，在复制法的 GC 中，需要

将所有的引用修改为指向被复制的新对象。但是在不允许修改对象的情况下，就不需要修改现有的引用，只要创建副本就能实现复制法（仅限于副本数不多的情况）。

GC 的实现

虽然实现 GC 要做的事情有很多，但是一次性完成太复杂的东西是会失败的，所以我打算分为以下步骤去实现。

Streem 基本上是把程序细分为任务单位来执行的，每次运行任务时都会进行 GC。

任务运行中如果有数据在任务之间传递（也就是调用了 `emit`），那么被传递的数据在该任务外也可能被引用，所以把这样的数据标记为“存活”。如果数据还用数组引用了其他对象，就也（递归地）标记这些被引用的对象。另外还（递归地）标记被全局变量引用的数据。Streem 的全局变量也是不能修改的，所以对象一旦被全局变量引用，它在整个程序执行期间就都是存活的。

一个任务运行结束后，除了被标记为“存活”的对象，将任务运行过程中其他被分配的对象全部删除，在下一个任务开始之前清除掉“存活”标记。

未来 GC 的实现

这次实现的只是最基本的 GC 功能，这项功能还有很大的改善余地。这里我们来聊一聊今后的设想。

Streem 的各个任务在不同的线程中运行，在多个线程访问同一个数据的情况下需要加锁等，这不但容易引发问题，还会影响运行性能。因此 Streem 为每个线程预先分配了一定的内存空间，创建数据时就从这块内存空间分配。由于这块内存空间不会被其他线程访问，所以不需要并发控制。

另外，用 `emit` 传递数据的任务在其他线程中运行时，要把数据递归地复制到该线程的内存空间中去。前面说过，Streem 的数据无法修改，基本不需要移动对象的内存空间，在这种情况下很容易实现复制法。

最终 Streem 的 GC 将会通过组合使用标记清除法和复制法来实现。另外，每当任务处理结束时对处理中创建的数据进行 GC，这也可以认为是分代 GC 的一种形式。

虽然还在设想当中，但是我觉得利用 Streem 语言的特性可以实现比之前的通用语言效率更高的 GC。我会继续研究下去的。

小结

关于 GC 的理论和实现的详细内容，请参考《垃圾回收的算法与实现》和《垃圾回收算法手册：自动内存管理的艺术》等相关图书，这些书也许会为你开启新世界的大门。

Streem 的语法对 GC 的影响让我觉得很有意思。在设计 Streem 的语法时，我只考虑了并行运

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

行，完全没有考虑 GC，没想到设计出来的语法竟在意想不到的地方产生了影响。

时光机专栏

语言的特性会影响GC

本节是 2016 年 2 月刊中刊登的内容。本节的主题是 GC，前半部分内容介绍了 GC 技术的概要，在有限的篇幅里我把相关的知识大致介绍了一遍（自卖自夸）。

但是有一点让我感到非常抱歉。在后半部分介绍 Streem 的 GC 时，我研究了 Streem 的语法是如何对 GC 产生影响的，并介绍了能够发挥 Streem 的特点的 GC 实现。内容本身并没有错，只是这个 Streem 专用的 GC 功能到现在还没有实现。

在 4-3 节的时光机专栏中我提到过自己干劲不足的事情，写这一节时我同样出现了这种情况。比起具体的 GC 实现，我希望大家在本节更多地关注探索某种语言的特点是如何影响 GC 功能的实现的这一思考过程。Streem 的 GC 早晚都要实现，但是我在写这篇稿子时还没有着手去做，希望在大家读到这本书的时候我已经实现了 GC 功能。

4-5 无锁算法

在并发编程中，并发控制对避免出现错误结果起到了非常重要的作用。然而，为对象加锁的代价是运行效率降低。为避免对性能产生影响而不向对象加锁的数据结构和算法称为“无锁”（lock free）。本节先从无锁的概念开始讲起，然后循序渐进，向大家介绍一下如何实现无锁。

首先我来讲一下在并发运行环境中线程按照预期运行（线程安全）的重要性。

我们以图 4-22 这种简单的队列实现为例。它的原理很简单，`struct queue` 这个数据结构拥有 `head` 和 `tail` 两个链接，通过向 `tail` 添加数据，从 `head` 取出数据，实现了一个先入先出（FIFO）队列。

```
#include <stdio.h>
#include <stdlib.h>

struct queue_node {
  void* v;
  struct queue_node* next;
};

struct queue {
  struct queue_node* head;
  struct queue_node* tail;
  //a
};

struct queue*
queue_new()
{
  struct queue* q;

  q = (struct queue*)malloc(sizeof(struct queue));
  if (q == NULL) {
    return NULL;
  }
  /* Sentinel node */
```

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

```
  q->head = (struct queue_node*)malloc(sizeof(struct queue_node));
  q->tail = q->head;
  q->head->next = NULL;
  //b
  return q;
}

int
queue_add(struct queue* q, void* v)
{
  struct queue_node *n;
  struct queue_node *node = (struct queue_node*)malloc(sizeof(struct queue_
node));

  node->v = v;
  node->next = NULL;
  //c
  q->tail->next = node;     ◄── (1)
  q->tail = node;           ◄── (2)
  //c
  return 1;
}

void*
queue_get(struct queue* q)
{
  struct queue_node *n;
  void *val;

  n = q->head;
  if (n->next == NULL) {
    return NULL;
  }
  //c
  q->head = n->next;
  val = (void*)n->next->v;
  //c
  free(n);
  return val;
}
```

图 4-22　队列的实现（第 1 版）

图 4-22 的程序提供了 3 个 API，如表 4-10 所示。API 与后面解说的版本相同。

表 4-10　队列的 API

API	功能
`struct queue* queue_new()`	创建队列
`int queue_add(struct queue* q, void* v)`	向队列添加
`void* queue_get(struct queue* q)`	从队列取出

并发运行的陷阱

即使是这种实现起来非常简单的队列，在使用了多个线程的并发运行环境中运行时也会有意想不到的问题发生。

在并发运行时，同一个数据有可能被同时使用、修改，这样就容易出现数据结构遭到破坏、数据丢失等情况，比如下面这种场景。请大家一边看图 4-22 的程序一边思考。

1. 线程 A 和线程 B 同时向队列写入数据
2. 线程 A 把新的节点链接到队列末尾（`q->tail`）节点的 `next`，并将 `q->tail` 修改为指向新的节点（图 4-22 的 (1) 和 (2) 部分）
3. 这时线程 B 也添加节点，时机不巧的话，线程 B 就会覆盖线程 A 修改的 `q->tail->next`
4. 之后，如果按照线程 A、B 的顺序修改 `q->tail`，线程 A 添加的节点就会丢失。如果反过来按照线程 B、A 的顺序修改，链接就失去了完整性，从 `head` 到 `tail` 的链接中断，在这之后向队列添加的元素将无法取出

这只是其中一个例子而已，还可能会发生其他各种各样的问题。而且这种问题时隐时现，可能运行 100 次才会失败 1 次，是一种很难发现的 bug。

这种类型的 bug 就是“海森堡 bug”。还有一种难以找出原因的 bug，即内存 bug，这两个 bug 都让开发者感到头疼。

引入并发控制

要避免这种问题，最简单的方法就是引入锁。在 POSIX 标准的 `pthread` 库中，用于并发控制的锁的数据类型定义如下。

```
pthread_mutex_t
```

`mutex` 是互斥（mutual exclusive）的缩写，使用这个 `mutex` 就可以进行队列操作的并发控制，修改起来很简单。

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

首先向 struct queue 结构体添加 pthread_mutex_t 类型的成员 lock（图 4-22 中 //a 的位置）。在初始化队列的 queue_new() 函数中，添加如下所示的锁的初始化处理（图 4-22 中 //b 的位置）。

```
pthread_mutex_init(&lock, NULL);
```

在 queue_add() 和 queue_get() 函数中，用以下语句将对象的使用、更新等需要进行并发控制的部分围起来（图 4-22 中 //c 的位置）。碰到处理途中用 return 等结束处理的情况时，不要忘记释放锁。

```
pthread_mutex_lock(&lock)
pthread_mutex_unlock(&lock)
```

其他部分与图 4-22 的代码相同，这里就不全部贴出来了。

为了保持数据的完整性，使用 mutex 保护不能同时访问的代码，可以使数据结构线程安全。一旦加了锁，在其他线程为了执行同一段代码再去加锁时，就会停止运行，等待锁的释放。这样一来，用 lock 和 unlock 围住的代码只能一次被一个线程执行，同时执行时出现的问题就不会再发生了。

锁的问题

使用 mutex 锁进行并发控制的优点是，不需要大规模修改原有代码即可支持多线程。

但是，在某个线程持有锁的这段时间里，其他线程只能等待锁被释放[①]。使用多线程编程进行并发运行的目的是尽可能地通过并行工作来提高性能，所以我们不希望看到线程因等待而停止工作这种情况。尤其是频繁访问的数据，并发控制的等待造成的时间损失让人觉得可惜。

而无锁就是一种不会产生等待时间，即使没有锁也可以实现并发运行的结构。

什么是无锁

无锁，顾名思义，就是不使用锁。也许有的读者会有疑问："不使用锁来进行并发控制，还能够处理好来自多个线程的访问吗？"

无锁的数据结构是通过"原子操作"这个神奇的技术来实现并发运行的。

这里的原子指的不是原子能。由于原子有不可分割的含义，所以用原子操作来称呼那些保证在

① 准确来说，还有一个 pthread_mutex_trylock() 函数可以用来判断代码是否已经被加锁，但光有这个函数还不能进行无锁处理，所以它并不能起到很好的效果。

处理过程中不会被线程调度机制打断的操作。

原子操作在执行过程中不会被打断，要么整个处理成功，要么由于前提条件不成立而处理失败。还不熟悉并发处理的人（其实我也不是非常熟悉）可能很难想象并发编程竟然连这一点都无法保证。

令人意外的是，计算机中不可分割的操作并不多。大家可能觉得 CPU 的一条指令是原子性的，但其实在现在的 CPU 中，机器码的一条指令常常被分割为多个 μ op 这种小的指令集，很多都是在经过优化之后再执行的。也就是说，即使是机器码的一条指令，严格来说也不一定是原子性的。这就意味着普通的方法是无法进行原子操作的。

CPU 拥有原子操作指令

然而原子性在并发环境中是非常重要的。线程操作库等（比如 mutex 的实现）使用的一定是原子操作。因此，现代几乎所有的 CPU 都准备了保证原子操作的指令，典型的原子操作指令有 CAS。

CAS 指令有 3 个参数。

```
CAS(a, b, c)
```

a 是地址，b 和 c 是整数值。上述代码表示如果 a 的地址的值是 b，就把值修改为 c 并返回真，否则返回假。也就是说，使用 CAS 可以保证从读取了某个地址的数据开始，到对值进行加工并写入新的值为止，在这期间不会有其他线程修改这个地址的值。

以前 C 语言是没有这样的指令的，但现在的 GCC（ver 4.1 之后的版本）增加了下面一条指令，由此，C 语言就可以使用 CAS 了。

```
__sync_bool_compare_and_swap()
```

另外，使用 CAS 指令可以构建无锁的数据结构。

无锁队列

下面我们就通过一个例子来看一下如何使用 CAS 指令实现无锁的数据结构。与图 4-22 的程序 API 兼容的无锁队列的实现如图 4-23 所示。

```
#include <stdio.h>
#include <stdlib.h>
```

```
struct queue_node {
  void* v;
  struct queue_node* next;
};

struct queue {
  struct queue_node* head;
  struct queue_node* tail;
};

struct queue*
queue_new()
{
  struct queue* q;

  q = (struct queue*)malloc(sizeof(struct queue));
  if (q == NULL) {
    return NULL;
  }
  /* Sentinel node */
  q->head = (struct queue_node*)malloc(sizeof(struct queue_node));
  q->tail = q->head;
  q->head->next = NULL;
  return q;
}

int
queue_add(struct queue* q, void* v)
{
  struct queue_node *n;
  struct queue_node *node = (struct queue_node*)malloc(sizeof(struct queue_
node));

  node->v = v;
  node->next = NULL;
  while (1) {
    /* 修改tail */
    n = q->tail;
    /* tail->next应该为NULL。如果为NULL就添加node，进入下一个处理 */
    if (__sync_bool_compare_and_swap(&(n->next), NULL, node)) {
      break;
```

```
    }
    /* 由于修改失败（被其他线程修改了），所以尝试修改q->tail */
    else {
      __sync_bool_compare_and_swap(&(q->tail), n, n->next);
    }
  }
  /* 修改q->tail */
  __sync_bool_compare_and_swap(&(q->tail), n, node);
  return 1;
}

void*
queue_get(struct queue* q)
{
  struct queue_node *n;
  void *val;

  while (1) {
    /* 取出q->head->next */
    n = q->head;
    /* 如果队列为空，返回NULL */
    if (n->next == NULL) {
      return NULL;
    }
    /* 修改q->head */
    if (__sync_bool_compare_and_swap(&(q->head), n, n->next)) {
      break;
    }
  }
  /* 取出值（从原来的头节点） */
  val = (void *) n->next->v;
  /* 释放不使用的节点（原来的头节点） */
  free(n);
  return val;
}
```

图 4-23　队列的实现（第 3 版）
使用了无锁的数据结构的例子

图 4-23 的程序中有几点值得注意。第一点是图 4-23 和图 4-22 的结构体定义完全相同，不需要向无锁队列添加锁等成员。

另一点是使用 CAS 修改结构体成员。前面提到过，使用 CAS 时需要指定待修改的地址的“现在的期望值”和“新值”。如果“现在的期望值”与该地址实际的内容不同，就说明值已经被其他线程修改，这时就要重新尝试修改。

我们来实际比较一下 queue_add() 和 queue_get() 的定义。图 4-22 的程序（由于没有考虑到处理过程中被其他线程修改的情况）只是通过赋值来进行修改，而图 4-23 的程序则通过重复使用 CAS 来尝试修改，直到修改成功为止。

保证处理顺序

5-2 节中我提到过，由于处理的先后顺序不是特别重要，所以整个程序只准备一个队列，在这个队列中添加处理数据的任务。

但实际按照这个结构进行测试后，我发现在处理比较简单的情况下程序能够正常运行，一旦情况稍微复杂一些，运行结果就会偏离预期。

出现这种情况的其中一个原因是需要考虑先后顺序的处理比预想的还要多。比如后面章节将介绍的 CSV 处理，它有一个功能：如果第一行都是字符串的字段，就把这行中的字段的值看作各字段的名称。但是，如果不能保证行的处理顺序，那么第一行就不一定会被最先处理了，不过第一行以外的行顺序发生改变倒没什么问题。

另外，在文件的读写方面，多数情况下我们不希望行的顺序发生变化。比如，在用 Streem 编写将读取的字符串转换为大写字母的过滤器时，我们就不希望行的先后顺序发生改变。

为了保证处理顺序，我决定修改队列的用法。具体来说，让每个流都拥有保存了顺序的数据队列，然后准备一个全局范围的队列，用于保存预定运行任务的流。

emit 之后，数据和处理数据的函数就会被添加到各个流的队列，等待处理的流就会被添加到全局队列（图 4-24）。

```
void
strm_emit(strm_stream* strm, strm_value data, strm_callback func)
{
  /* data不为nil时添加到queue */
  if (!strm_nil_p(data)) {
    /* dst：下一个流 */
    strm_stream* dst = strm->dst;
    /* 向下一个流的队列添加保存了函数和data的strm_task */
    strm_queue_add(dst->queue, strm_task_new(dst->start_func, data));
    /* 为了预定在下一个流运行任务，把下一个流添加到队列 */
    strm_queue_add(queue, dst);
  }
  /* 如果指定了func，就把它也添加到队列 */
```

```
    if (func) {
      /* 这里data没有意义，指定为nil */
      strm_queue_add(strm->queue, strm_task_new(func, strm_nil_value()));
      /* 向优先级较低的队列（prod_queue）添加流 */
      strm_queue_add(prod_queue, strm);
    }
}
```

图 4-24 使用了新队列的 emit

总的来说，strm_emit() 是按照下列顺序对下一个流 emit 数据的。

1. 向下一个流的队列添加任务（函数和数据组）
2. 把下一个流添加到全局队列，预定运行任务
3. 在指定了 func 时，向自己的流的队列添加任务，向优先级较低的队列添加流，预定运行任务

之所以需要优先级较低的队列，是因为如果过于频繁地运行生产者，队列就会变得很长，造成内存空间浪费，所以需要降低运行生产者的优先级。

将来我打算尝试用其他方法来控制数据量。

并发控制

并发控制是在实际运行中出现的另一个问题。属于同一个流的多个任务在同时运行时可能会争夺数据，这是因为各个任务不是线程安全的。

前面的实现都是通过固定运行流的工作线程来使同一个流的任务不能同时运行的，但是从有效使用 CPU 内核的观点来看，固定工作线程并不是一个好方法。

于是我对各工作线程的处理循环的实现做了比较大的修改（图 4-25）。

```
static void*
task_loop(void *data)
{
  strm_stream* strm;

  for (;;) {
    /* 从队列中取出流 */
    strm = strm_queue_get(queue);
    /* 如果队列为空，就从优先级较低的队列中取出流 */
    if (!strm) {
      strm = strm_queue_get(prod_queue);
```

```
    }
    if (strm) {
      /* 用strm_excl作为标志位进行并发控制 */
      if (strm_atomic_cas(strm->excl, 0, 1)) {
        struct strm_task* t;

        /* 对各个流的队列中存在的任务全部进行处理 */
        while ((t = strm_queue_get(strm->queue)) != NULL) {
          /* 运行任务 */
          task_exec(strm, t);
        }
        /* 恢复标志位 */
        strm_atomic_cas(strm->excl, 1, 0);
      }
    }
    /* 如果流全部被关闭，就结束循环 */
    if (stream_count == 0) {
      break;
    }
  }
  return NULL;
}
```

图 4-25　修改后的工作线程的处理循环

处理循环的步骤如下所示。

1. 从队列中取出流
2. 如果流中没有任务正在运行，就设置标志位（strm_excl），然后运行所有积攒的任务
3. 如果流中有任务正在运行（已设置标志位），则跳过

在多个任务被添加到队列时，会出现因为设置了标志位而导致流被忽略的情况，但这一处理的目的不过是安排任务的执行，所以也没有什么问题。设置了标志位就意味着对流的处理正在运行，既然是正在运行，那么积攒在队列中的所有任务就都会被执行。

话虽如此，但效率不高也是事实，我们今后再去研究更好的做法。

图 4-25 的函数调用了下面的函数。

```
strm_atomic_cas()
```

这是调用下面这个 GCC 的函数的宏。

```
_ _sync_bool_compare_and_swap()
```

我自己老是记不住这个扩展命令，所以想出了这个办法。

图 4-25 中使用 strm_atomic_cas() 的部分是线程安全的标志位的典型用法示例。

无锁运算

GCC 4.1 之后提供的原子运算就不只是 CAS 了。Streem 准备了把这些运算汇总起来的 atomic.h 头文件（图 4-26）。现在只准备了使用 GCC 扩展命令的宏，将来如果需要支持其他编译器（比如 Visual C++），只需修改这个头文件即可。

```
#define strm_atomic_cas(a,b,c) __sync_bool_compare_and_swap(&(a),(b),(c))
#define strm_atomic_add(a,b) __sync_fetch_and_add(&(a),(b))
#define strm_atomic_sub(a,b) __sync_fetch_and_sub(&(a),(b))
#define strm_atomic_inc(a) __sync_fetch_and_add(&(a),1)
#define strm_atomic_dec(a) __sync_fetch_and_sub(&(a),1)
#define strm_atomic_or(a,b) __sync_fetch_and_or(&(a),(b))
#define strm_atomic_and(a,b) __sync_fetch_and_and(&(a),(b))
```

图 4-26　atomic.h

我费了很大功夫才实现无锁运算，所以我也把它用在了 Streem 语言处理器的其他地方。

首先，在增减变量 stream_count 时使用了 strm_atomic_inc() 和 strm_atomic_dec()。stream_count 用于统计活跃的流的数量，以检查流处理是否已结束。由于普通的增量运算符不保证原子性，所以在更新时机不对时可能无法正确地计数。但如果是 strm_atomic 系列的函数，就不用担心这个问题了。

stream_count 为 0 就意味着已经没有要处理的流了，这时可以放心地结束事件循环。

在维护用于流的存活管理的被引用数时，也使用了 strm_atomic_inc 和 dec。在“流的混合”等一个流被多个上游的流引用的情况下，如果所有上游的流都运行结束了，就可以“关闭”下游的流，因此需要统计每个流被多少个流引用了。每当流被连接到一起时，被引用数就会增加，当上游流被关闭时，被引用数减少，当被引用数变为 0 时，就关闭流。

另外，我想这项技术也可以用在统计流的元素数的 count() 中。不过目前任务处理进行了并发控制，还不需要特意修改为原子操作，所以我还没有动手去改。

小结

原子操作指令及其相关的数据结构对有效使用多核很有效果。本节讲解了无锁数据结构的原理和实现。

时光机专栏

无锁算法的实现有问题

本节是 2016 年 5 月刊中刊登的内容。关于这一节的内容，我也表示非常抱歉。无锁算法的介绍并没有问题，但这次讲解的代码却有几个问题，所以我不建议大家把这一节的代码用在自己的项目中。

第一个问题是 ABA 问题。本节使用的无锁算法为了判断操作过程中数据是否已被修改，采取了以下做法：如果使用 CAS 检测出当前的值与本次操作前的值不同，就认为在操作过程中数据被其他线程修改过，要重新进行处理，具体过程如下所示。

1. 线程 1：保存旧值，开始修改处理
2. 线程 2：在线程 1 还在处理的过程中修改数据，完成处理
3. 线程 1：检查旧值是否发生了变化，如果已变化，则重新进行处理

可是，如果线程 2 修改的旧值在 `free()` 之后又通过 `malloc()` 被重新使用，那么修改的值就有可能和旧值相同。虽然这种可能性很低，但并不代表没有。在这种情况下，无锁算法就无法检测出数据是否被其他线程修改过，这就是 ABA 问题。发生 ABA 问题时，数据的完整性可能会遭到破坏。ABA 问题在无锁栈上频繁发生，在队列上也会发生。

解决这个问题的方法有几种，其中一种是使用名为风险指针（hazard pointer）的数据结构。风险指针用于延迟释放还在用的值，在值被引用时，地址不能被重复利用，所以值不会发生冲突。另一种做法是使用 GC。GC 不会重复利用还在使用的指针，所以也可以避开 ABA 问题。

现在还不能确定是否是 ABA 问题导致了无锁算法的实现出现问题，但可以确定的是本节介绍的代码在高负荷运行时会发生数据遗漏的情况。这也是不建议大家使用本节代码的原因。

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

第 5 章

强化流编程

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

5-1 管道编程

前面几节介绍的都是语言处理器的实现的相关内容，从这一节开始我们来聊聊如何强化库。本节将探讨如何用Streem实现CSV数据的处理和打方块游戏。由于需要进行状态管理，所以我们也要实现“内置数据库”。

管道编程是 Streem 的一个特点，它与普通的编程方式有很大不同。以前我简单介绍过管道编程的方式，这次我们来更加具体地探讨一下如何用管道去开发各种类型的处理，也许会让你对管道编程有一个新的认识。

我在前面介绍过构成任务管道的典型模式，当时介绍了以下几种。

- 生产者 – 消费者模式
- 轮询调度模式
- 广播模式
- 汇总模式
- 请求 – 应答模式

当时没有深入讲解具体的程序，所以大家可能印象不深，这次我们就来深入了解一下。

数据统计

Streem 最擅长的就是进行统计处理。读入数据、选择符合条件的数据、加工数据以及统计数据等都可以轻松地用管道来编写。

下面我们以 CSV 数据为对象进行一些简单的统计处理。

假设我们有如图 5-1 所示的 CSV 数据，其中列出了各种编程语言及其发布年份和作者。

```
year,name,designer
1995,Ruby,Yukihiro Matsumoto
1987,Perl,Larry Wall
1991,Python,Guido van Rossum
1995,PHP,Rasmus Lerdorf
1959,LISP,John McCarthy
1972,Smalltalk,Alan Kay
1990,Haskell,Simon Peyton Jones
1995,JavaScript,Brendan Eich
1995,Java,James Gosling
1993,Lua,Roberto Ierusalimschy
1987,Erlang,Joe Armstrong
2007,Clojure,Rich Hickey
2003,Scala,Martin Odersky
2012,Elixir,José Valim
```

图 5-1　编程语言列表

我们试着从这个列表中选出在 21 世纪发布的语言，相应的 Streem 程序如图 5-2 所示。

在这个列表中，诞生于 21 世纪的语言有 Clojure、

Scala 和 Elixir 这 3 种。虽然我很想把 Streem 加到这个列表里，但 Streem 还没有达到实用的程度，所以我这么想就有些狂妄自大了。

```
fread("lang.csv")|csv()|filter{x->x.year>2000}|stdout
```

图 5-2　选出 21 世纪诞生的语言的 Streem 程序

那么，20 世纪诞生的语言有多少种呢？这个列表包含的数据并不多，我们自己也能数得过来，但如果数据量很大，就需要通过程序进行统计了。统计元素的个数时使用 count() 函数（图 5-3）。

```
fread("lang.csv")|csv()|filter{x->x.year<2001}|count()|stdout
```

图 5-3　对 20 世纪诞生的语言的数量进行统计的 Streem 程序

在这个列表中，诞生于 20 世纪的语言有 11 种。果然，Streem 很擅长进行这样的处理。

Web 服务与管道

不过，Streem 并不是专门用来统计数据的语言。管道编程也可以用于 CSV 统计以外的场景，比如可以用在 Web 技术上。

现在的很多软件都是使用 Web 技术开发的，通过 CD 和 DVD 等媒介在计算机上安装的软件已经很少了，大多数软件要通过浏览器访问，就连商用软件也都是使用 Web 技术开发的。

而在智能手机中，我们却很少通过浏览器使用软件[①]，一般都是去安装应用。不过这些应用一般也是使用 HTTP 协议从服务器端的 API 获取信息的。

因此，不管是在手头的设备上安装应用，还是通过浏览器访问服务器端软件，现在基本上都离不开 Web 技术。

HTTP 的软件构成

以 HTTP 为中心的软件构成如图 5-4 所示。

首先，客户端向服务器端发送请求。服务器端根据请求进行处理，然后把结果返回给客户端。虽然从严格意义上讲还有很多特殊的情况，但 HTTP 基本上就是这种简单的请求 – 应答模式。

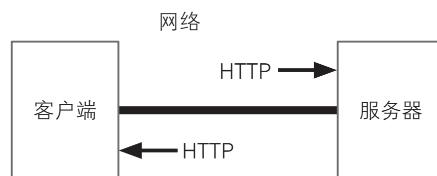


图 5-4　HTTP 的软件构成

① 最早的 iPhone 里还没有应用，所有软件都需要通过浏览器访问。应用出现以后，智能手机就发生了与计算机完全相反的变化，很有意思。

表 5-1 中列出了 HTTP 的请求类型。

表 5-1　HTTP 的请求类型

类型	含义	用法
GET	获取	获取页面信息
PUT	修改	文件上传等
POST	创建	信息在请求体部分的表单里
DELETE	删除	删除资源
HEAD	请求头	只获取请求头信息

在考虑通过 HTTP 访问 API 的情况下，理想的做法是把对服务器的请求当成数据库操作，然后按照表 5-2 的方式进行。我们把对数据库的 CREATE、READ、UPDATE 和 DELETE 操作简称为 CRUD。

表 5-2　与数据库操作相对应 HTTP 请求

数据库操作	含义	请求类型
CREATE	创建	POST
READ	获取	GET
UPDATE	修改	PUT
DELETE	删除	DELETE

服务器端架构

在图 5-4 的构成 HTTP 的软件中，我们重点看一下服务器端软件，服务器端软件的构成如图 5-5 所示。客户端的连接管理、HTTP 请求的语法分析、进程管理等主要由 HTTP 服务器负责。服务器端软件拿到解析好的信息之后开始进行处理，然后将结果返回给 HTTP 服务器。之后，HTTP 服务器拼装 HTTP 应答，并将其送还给客户端。

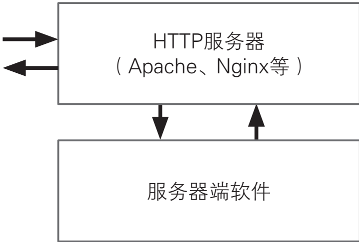


图 5-5　服务器端软件的构成

也就是说，从服务器端程序的层面来看，整个处理的过程是：收到 HTTP 请求的信息之后对数据进行加工，然后把结果作为 HTTP 应答返回。这个处理也可以用管道来编写。

我们以一个简单的 ToDo 程序作为示例。认证等由 HTTP 服务器执行，服务器端软件负责接收“显示”“创建”“修改”“删除”这 4 个处理。

这些处理依照前面介绍的 CRUD 规则，分别按 GET、POST、PUT 和 DELETE 这 4 种请求类型接收，代码如图 5-6 所示。图 5-6 虽然是 Streem 的代码，但由于支持 HTTP 的库还没有完成，所以这只能算是预想中的代码。

render() 函数用于创建向客户端发送的数据，不过图 5-6 中没有显示 render() 函数的内容。大家可以把它想象为一个根据状态（成功或失败）和用户 ID 来创建应该显示的内容的 HTML（API 的情况下就是 JSON）的函数。

render() 创建的结果（用于显示的数据）会传给 http_response()，然后再从这里发送给 HTTP 服务器。

图 5-6 的程序虽然是伪代码[①]，但是通过这个例子，相信大家也掌握了一些使用管道编写普通程序的方法。

① Streem 的周边库还没有达到实用的程度，对此我感到非常抱歉。

```
db = kvs()
http_request() | map{req ->
  if (req.type == "GET") {          # 显示
    emit :ok
  }
  else if (req.type == "POST") {    # 创建
    db.put(req.todo_id, [req.title, req.due])
    emit :ok
  }
  else if (req.type == "PUT") {     # 修改
    db.put(req.todo_id, [req.title, req.due])
    emit :ok
  }
  else if (req.type == "DELETE") { # 删除
    db.put(req.todo_id, nil)
    emit [:ok, req.user]
  }
  else {                           # 其他（错误）
    emit :error
  }
} | render() | http_response()
```

图 5-6　使用 Streem 开发的服务器端软件

电子游戏的示例

我们再来考虑一个看起来很难用管道编写的例子，比如编写一个打方块那样的电子游戏。

遗憾的是，现在还不能用 Streem 开发电子游戏，所以我们假定实时键盘输入和图形输出的库已经存在，以此来继续我们的话题。

那么如何用管道编写电子游戏呢？方法有很多。这次我想到的方法是准备 3 个管道，具体如图 5-7 所示。

```
# 更新频率（每秒30次）
fps = 30
# 更新间隔
tick = 1/fps*1000
# 保存游戏状态的内存数据库
board = kvs()
```

```
kvs.put(:paddle_x, 0)
kvs.put(:ball_x, 0)
kvs.put(:ball_y, 0)
kvs.put(:ball_x_vec, 1)
kvs.put(:ball_y_vec, 1)
kvs.put(:num_balls, 5)

key_event() | each{x ->
  if (x == "LEFT") {
    board.update(:paddle_x, {x -> x-1})
  }
  else (if x == "RIGHT") {
    board.update(:paddle_x, {x -> x+1})
  }
}

timer_tick(tick) | each{x ->
  # 更新球的位置
  x_vec = board.get(:ball_x_vec)
  y_vec = board.get(:ball_y_vec)
  x = board.update(:ball_x, {x -> x + x_vec})
  y = board.update(:ball_y, {y -> y + y_vec})

  # 碰撞检测
  if (x == 0) {  # 球到最下面了
    if (ball_hit_paddle()) {
      board.update(:ball_x_vec, {x -> -x})
      board.update(:ball_y_vec, {y -> -y})
    }
    else {  # 掉下去了
      n = board.update(:num_balls, {n -> n-1})
      if n == 0 {
        game_over()
      }
    }
  }
  else if (ball_hit_block()) {
    erase_block()
  }
}
```

```
timer_tick(tick) | each{x ->
  # 图形输出
  display_board()
}
```

图 5-7 打方块

第 1 个管道接收用户的输入，更新玩家（球和托球的板）的位置。

第 2 个管道定期被调用，更新各个物体（球的位置和方块的状态）。

第 3 个管道把更新结果输出为图形。

通过像这样分割为几个管道来编写，可以使每个管道的处理都非常简单，易于理解，也更容易维护。

这个实现的关键点在于把处理分为多个管道，以及把游戏的状态保存到内存数据库。

不可变与状态

Streem 的一大特点是数据结构不可变（不能修改），这个特点给程序的编写带来了不少麻烦。

基本不带副作用的纯函数式语言 Haskell 使用“单子”（monad）结构来管理状态变更等有副作用的处理，不过老实说，并不好用。

数据结构同样不可变的 Erlang 是以 Actor 模型为基础的语言，它使用两种方法来管理状态。一种方法是把状态封闭在独立运行的进程里，通过与进程相互收发消息来进行状态的修改和读取。这种方法虽然巧妙，但不适合 Streem 这种没有显式的进程（和线程）的语言。

另一种方法是使用 ETS 和 Mnesia 等内置在 Erlang 语言处理器中的数据库。也就是说，即使普通的数据结构不可变，数据库这种专门用来保存状态的数据结构也是可以修改的。

这对于习惯了普通编程的人来说也非常容易理解。实际上本节的 ToDo 程序和打方块程序就是使用数据库来实现可修改的状态的。

引入嵌入式数据库的 kvs

考虑到这些，就需要在 Streem 中引入嵌入式数据库了。Clojure 语言的数据结构在原则上也是不可变的，但该语言使用 STM（Software Transactional Memory，软件事务内存）的方式实现了可修改的状态。事务是维持数据完整性的方法，Clojure 就使用事务来维持数据结构的完整性。

虽然 Streem 处理器引入的嵌入式数据库的细节部分还有待商榷，不过我还是先开发了一个名为 `kvs` 的简易数据库。

`kvs` 是一个简单的键值存储库（key value store），其功能如表 5-3 所示。

这里只需要对 `update` 的行为加以说明。`update` 把键和函数当成参数。函数把旧值当成参数

接收，它的返回值会成为新值。

然后 tnx 函数开始事务处理，执行作为参数接收到的函数。这个函数接收表示事务的对象。事务与数据库的行为相同，事务结束时，对事务的修改会更新到数据库。如果在事务开始之后修改了数据库中的数据，从而产生了冲突，那么就放弃前面的修改，从头开始执行事务函数。如果在事务函数运行过程中发生了错误，那么就放弃事务中到此为止的修改，然后再次抛出错误。

表 5-3　kvs 的功能

API	作用
db = kvs()	打开内存数据库
db.put(key, val)	设置数据
db.get(key)	获取数据
db.update(key, f)	修改数据
db.txn(f)	事务
db.close()	关闭数据库

kvs 的实现

接下来我们就开始实现 kvs。我强调过很多次，让实现动起来是最重要的。这里先不考虑性能等问题，我们要尽快让实现满足运行的需求。当然，这个实现在实际使用过程中会暴露一些性能问题，存在需要改进的地方。到那时我们再细致地测试一番，在明确问题出现的原因之后进行改善。设计、实现、测试和改善是软件开发的金规铁律。

我想尽快实现 kvs。幸运的是，Streem 在实现中采用了内存上的键值存储库 khash（虽然只是散列表），我就暂时用它来实现 kvs。

khash

khash 是 C 语言使用的散列表库，以 MIT 许可证提供。khash 最大的特点是仅由头文件组成。使用 khash 时要包含（include）khash.h 头文件，使用宏来声明要使用的散列表。访问用的函数等也通过宏来定义。

比如可以像图 5-8 那样，定义一个键为 Streem 的字符串、值为任意 Streem 数据的 kvs 的散列表。KHASH_INIT 是定义散列表的宏，这个宏有 6 个参数，其含义如表 5-4 所示。

```
#include "strm.h"
#include "khash.h"

KHASH_INIT(kvs, strm_string, strm_value, 1,
kh_int64_hash_func, kh_int64_hash_equal);
```

图 5-8　kvs 的散列表的定义

表 5-4 KHASH_INIT 的参数

顺序	名称	含义
1	name	散列表名
2	khkey_t	键的类型
3	khval_t	值的类型
4	kh_is_map	map（1）、set（0）
5	hash_func	键的散列函数
6	hash_equal	键的比较函数

散列表名是散列表结构体的名称。散列表通过“khash_t（名称）”这一类型来访问。访问图 5-8 定义的散列表的程序如图 5-9 所示。

```
int
main()
{
  int ret, is_missing;
  khiter_t k;
  strm_string key = strm_str_intern("foo", 3);
  strm_string key2 = strm_str_intern("bar", 3);
  khash_t(kvs) *h = kh_init(kvs);

  /* 使用kh_put获取保存位置 */
  k = kh_put(kvs, h, key, &ret);
  /* 通过向kh_value()赋值来实际保存数据 */
  kh_value(h, k) = strm_int_value(10);
  /* 使用kh_get获取访问位置 */
  k = kh_get(kvs, h, key2);
  /* 如果没有值，k为kh_end（最后的位置） */
  /* 如果有值，使用kh_value()获取值 */
  is_missing = (k == kh_end(h));
  /* 通过循环kh_begin()到kh_end() */
  /* 可以查找所有的元素。由于也有空的位置 */
  /* 在访问元素之前需要用kh_exist()检查 */
  for (k = kh_begin(h); k != kh_end(h); k++)
    if (kh_exist(h, k))
      kh_value(h, k) = strm_int_value(1);
  /* 使用kh_destroy()销毁散列表 */
  kh_destroy(kvs, h);
  return 0;
}
```

图 5-9 访问散列表

使用 khash 时非常方便，只需引用头文件即可。如果大家有机会用 C 语言开发程序，建议使用这个库。不过它的 API 比较特殊，需要花一些时间去熟悉。

使用 khash 实现 kvs

讲到这里，（如果是最简单的实现的话）开发键值存储库就很简单了。图 5-10 摘录了 kvs 实现的最初的核心代码。

ns 成员保存了用 strm_state 结构体表示的命名空间，用来代替面向对象语言的类。我们只要注意到这一点，就不会觉得内容太难了。

```
struct strm_kvs {
  STRM_PTR_HEADER;
  khash_t(kvs) *kv;
};

static khash_t(kvs)*
get_kvs(int argc, strm_value* args)
{
  struct strm_kvs *k;

  if (argc == 0) return NULL;
  k = (struct strm_kvs*)strm_value_ptr(args[0], STRM_PTR_KVS);
  return k->kv;
}

static int
kvs_get(strm_task* task, int argc, strm_value* args, strm_value* ret) {
  khash_t(kvs)* kv = get_kvs(argc, args);
  strm_string key = strm_str_intern_str(strm_to_str(args[1]));
  khiter_t i;

  i = kh_get(kvs, kv, key);
  if (i == kh_end(kv)) {
    *ret = strm_nil_value();
  }
  else {
    *ret = kh_value(kv, i);
  }
  return STRM_OK;
}
```

```
static int
kvs_close(strm_task* task, int argc, strm_value* args, strm_value* ret) {
  khash_t(kvs)* kv = get_kvs(argc, args);
  kh_destroy(kvs, kv);
  return STRM_OK;
}

static strm_state* kvs_ns;

static int
kvs_new(strm_task* task, int argc, strm_value* args, strm_value* ret) {
  struct strm_kvs *k = malloc(sizeof(struct strm_kvs));

  if (!k) return STRM_NG;
  k->ns = kvs_ns;
  k->type = STRM_PTR_KVS;
  k->kv = kh_init(kvs);
  *ret = strm_ptr_value(k);
  return STRM_OK;
}

void
strm_kvs_init(strm_state* state) {
  kvs_ns = strm_ns_new(NULL);
  strm_var_def(kvs_ns, "get", strm_cfunc_value(kvs_get));
  strm_var_def(kvs_ns, "put", strm_cfunc_value(kvs_put));
  strm_var_def(kvs_ns, "update", strm_cfunc_value(kvs_update));
  strm_var_def(kvs_ns, "txn", strm_cfunc_value(kvs_txn));
  strm_var_def(kvs_ns, "close", strm_cfunc_value(kvs_close));
  strm_var_def(state, "kvs", strm_cfunc_value(kvs_new));
}
```

图 5-10 kvs 的早期实现（摘录）

并发控制

但是这种简单的实现还不够实用，因为还没有在多线程、多任务的 Streem 语言中进行必不可少的并发控制。

比如在修改散列表的过程中，如果有其他线程尝试读取数据，可能就会读出不存在的数据。如

果在写入数据时发生冲突，数据结构就可能遭到破坏。

要避免这样的问题，就需要进行并发控制。方法有很多种，这次我使用了锁和事务来进行并发控制。

使用锁进行并发控制

使用锁进行并发控制的方法比较简单。为每个散列表准备一个 pthread_mutex_t 锁，在访问共享的数据结构（这次是散列表）的代码前后，用下面的函数围起来。

```
pthread_mutex_lock()
pthread_mutex_unlock()
```

这样做至少可以避免数据被破坏。

事务的实现

但是对散列表这样的数据库来说，仅仅通过锁来保护数据不被破坏，有时候是不够的。

这是因为在多个线程同时写入时，即使数据库本身没有损坏，有时也无法保持数据的完整性。比如，在从银行账户 A 向账户 B 汇款的过程中，如果因为正好有另一笔汇款在进行而导致本应汇到账户 B 的钱消失了，那就成了大问题。

为了避免这样的情况出现，数据库采用了事务这一方法。事务可以保证操作结果为以下 3 种情况：成功修改了一系列的状态；发生冲突时重新操作；失败之后将状态恢复为事务开始前的状态。

kvs 使用 txn 函数实现了事务。txn 函数开始事务，把事务对象作为参数传递给作为参数传来的函数。事务对象与普通的 kvs 数据库的行为相同，在事务结束时，（如果操作成功）向数据库写入数据。

事务的实现非常复杂，由于篇幅的关系，这里不再详细讲述，请读者参考 github.com/matz/streem 上的 src/kvs.c 源代码。

小结

本节介绍了 Streem 的管道编程，另外还实现了对状态变化的管理来说非常重要的 kvs 数据库。Streem 就是这样一点一点地变实用的，这也是连载和语言开发同时进行的乐趣所在。

时光机专栏

应该有的功能还是要实现的

本节是 2016 年 3 月刊中刊登的内容，探讨了如何用 Streem 特色之一的管道编程来编写代码。

Streem 也好，极大地影响了 Streem 的函数式语言也好，都存在可以发挥它们特长的领域，在这些领域中能够优雅地编写程序。介绍这些语言的书也喜欢拿这些领域的问题作例子，解释如何用这些语言优雅地解决问题。

但是我们开发者平时遇到的很多问题都不能优雅地解决。不对，是能够解决，但是否优雅就是另外一回事了。不习惯的人会觉得很多问题都不能直接解决，可要是每解决一种类型的问题就换一门语言，那成本（主要是精神上的）就太高了。

相比较而言，Streem 不是通用语言，而是管道编程专用语言，所以可以不用那么担心。但即便如此，一门语言能做的也肯定是越多越好。

于是，本节以 Streem 可以开发各种程序为前提，探讨了很多内容。探讨的结果就是使用内存数据库 `kvs`。引入数据库的目的是让变量可以在语法层面进行修改，从而使实现变简单，但这么做又显得有些小题大做。不过看过 Clojure 的尝试之后，我觉得这种做法也是可以接受的。虽然这次没有考虑性能，但我想本次的实现在一定程度上也体现了这一理念的有效性。

关于在本节的示例代码中出现的函数群，除了 `kvs` 之外，都是还没有实际提供的函数，比如 `http_request()`、`key_event()` 和 `timer_tick()` 等。我发现要想用 Streem 开发出实用的程序，就需要大量提供这种工具函数。在今后的连载中，我将继续探讨和实现这些函数。

5-2 管道的构成要素

5-1节进行了管道编程的实践，开发了管道编程的重要构成要素kvs。本节将继续探讨管道编程的实践所需要的构成要素，并集齐必要的工具。在整理完概念之后，我将大幅修改实现的代码。

Streem 开发到现在，我注意到一个问题，就是当初临时起的一些名字与其实际代表的东西并不相符。

比如 Streem 中有一个很重要的结构体，名叫 `strm_task`，但是在 Streem 的语言层面并不存在“任务”这个概念。这个“任务”到底表示什么，我本人也有点记不清了。实现上的概念和语言上的概念发生了偏离，怎么看都觉得不协调。

这几个月我一直在想这个不协调的问题。为了彻底搞清楚管道编程，我想先对概念进行一次整理。

管道的构成要素

我们再次对 Streem 管道中的构成要素进行梳理，重要的是以下 4 个概念。

- 管道
- 流
- 任务
- worker

其中最重要的概念是流。流是促成数据流转的处理。流中包含产生数据的生产者、接收数据并对其进行加工的过滤器，以及接收数据进行消费的消费者 3 种角色。作为编程模型，流结合在一起之后还是流，但是在语言处理器的层面上，则用 `strm_stream` 结构体表示一个处理。

从生产者开始到消费者为止，被串起来的流称为管道。由于 Streem 的语言处理器中（目前）没有处理管道的部分，所以它还只是停留在概念上。

负责处理流中的一份份数据的是任务。Streem 处理器分别用一个 C 函数来表示任务的处理。每当流向下一个阶段 `emit` 数据时，就把任务结构体添加到运行队列，预定将来运行这个任务。

worker 指的是运行任务的线程。Streem 处理器会生成与计算机 CPU 内核数数量相同的 worker。worker 从队列的前面开始依次取出任务并运行。

全局范围重命名

表 5-5　修改名称

种类	原来的名称	修改后的名称
类型	strm_task	strm_stream
变量	task	strm
结构体	strm_thread	strm_worker

既然概念已经整理完毕，接下来就开始对源代码进行相应的修改吧。我修改了 Streem 代码的结构体名、函数名和变量名等，具体内容如表 5-5 所示。

实际上在 core.c 和 queue.c 实现的事件处理代码中，有一个名为 strm_queue_task 的结构体，它与 strm_task 的名称相似，但二者毫无关系，这个结构体就相当于这次概念整理后的任务。这部分的代码也被大幅修改，我会在后面加以讲解。

一般来说，大规模修改源代码中出现的名称，会导致软件丧失兼容性。尤其是开源软件，改名伴随的风险非常大。

Ruby 在开发早期也大规模地修改过名称。当时为了避免与其他库的名称重复，就在 Ruby 的名称中统一加上了“rb_”前缀。这是将近二十年前的事了。尽管当时的用户很少，但还是接连出现了库不能正常工作等情况，给我留下了一段辛酸的回忆。

至于 Streem，由于它缺乏实用性，几乎没有人使用，而且也没有提供扩展用的 API，所以可以随心所欲地修改名称。为了使实现变得更好，今后我可能还会大幅度地对名称进行修改。

增加调试用的 strm_p()

我们顺便再增加一个功能。

引入 NaN Boxing 之后，Streem 的对象就都可以用 64 位整数表示了。虽然 NaN Boxing 的运行效率很高，但由于看不到内部情况，调试时非常麻烦。因此，我在调试器中增加了显示对象内部情况的 strm_p() 函数。

strm_p() 的实现很简单（图 5-11）。只需要使用 strm_to_str() 函数将作为参数传过来的对象转换为字符串对象，然后用 strm_str_cstr() 函数取出 C 的字符串指针，最后用 fputs() 显示到标准输出即可。

```
strm_value
strm_p(strm_value val)
{
  char buf[7];
  strm_string str = strm_to_str(val);
  const char* p = strm_str_cstr(str, buf);
  fputs(p, stdout);
  fputs("\n", stdout);
  return val;
}
```

图 5-11　显示对象内部情况的 strm_p() 函数

这里需要稍微说明一下 strm_str_cstr() 函数的 buf 参数。由于 Streem 的 NaN Boxing 会把 6 个字符以下的短字符串直接保存在 strm_value 中，所以无法取出字符串指针。因此，在复制这样的字符串内容时就需要 buf 空间。由于保存的字符串最大为 6 个字符（字节），加上末尾的 NUL 占用的空间，所以 buf 最少需要 7 字节大小。

另外，我还准备了能够为每个对象定制字符串表示的结构。如果设置了命名空间的对象持有 to_str() 函数，它的返回值就会被用作对象的字符串表示。

管道编程的模式

在 5-1 节开发的管道编程有以下 3 种模式。

- 生产者生产的数据经过过滤器加工，由消费者输出
- 生产者生产的数据经过过滤器加工，写入数据库（kvs）
- 每隔一段时间就启动一次处理，输出从数据库获取的数据

前面已经出现的生产者和消费者分别如表 5-6、表 5-7 所示。

另外，表 5-8 列举了已出现的和本次新增加的过滤器。当然这些过滤器能做的事情有限，今后我还要继续增加过滤器。

新增加的过滤器里有一个 reduce()。reduce() 是对流的元素进行归约的函数，调用方式如下所示。

```
reduce(b){x,y->...}
```

当上游的流传来元素（e1, e2, ...）时，首先将 b 作为第 1 参数，将 e1 作为第 2 参数来对函数进行调用。如果令其结果为 r1，那么对下一个元素 e2 进行处理时，就将 r1 作为第 1 参数，将 e2 作为第 2 参数来对函数进行调用。后面的元素也要进行相同的处理，最后得到的结果会作为输出传给下游的流。

表 5-6　Streem 的生产者

名称	作用
stdin	标准输入
fread()	读取文件
tcp_server()	套接字连接
tcp_socket()	读取套接字
seq()	一定范围的整数
rand()	随机数序列
tick()	每隔一定时间产生一次事件

表 5-7　Streem 的消费者

名称	作用
stdout	标准输出
stderr	标准错误输出
fwrite()	写入文件
tcp_socket()	写入套接字
each()	循环

表 5-8　Streem 的过滤器

函数名	作用
map()	用函数执行结果进行替换
filter()	选择符合条件的数据
count()	提供元素数量
sum()	计算元素的总和
csv()	将 csv 字符串转换为数组
flatmap()	展开数组的 map（新增加）
split()	分割字符串（新增加）
reduce()	归约（新增加）
reduce_by_key()	根据键进行归约（新增加）

使用 reduce() 进行阶乘计算的代码如图 5-12 所示。其他语言常使用递归和循环进行阶乘计算，与这些语言的代码相比，使用 reduce() 进行阶乘计算的代码看起来有很大的不同。

```
seq(6)|reduce{x,y->x*y}|stdout
```

图 5-12 使用 reduce() 进行阶乘计算的代码

另外，已经出现的 sum() 函数能够使用 reduce() 像图 5-13 那样进行定义。如此定义的 sum() 会返回过滤器流，把从上游接收的数据的总和传给下游。

```
def sum() {
  reduce(0) {x,y -> x+y}
}
```

图 5-13 使用 reduce() 对 sum() 进行定义

使用归约进行单词计数

我们还需要 reduce 的变体，即根据键进行统计的 reduce_by_key() 函数，这个函数用在可以说是管道编程界的 Hello World 的单词计数中。使用 Streem 进行单词计数的程序如图 5-14 所示。

```
stdin
| flatmap{s->s.split(" ")}
| map{x->[x, 1]}
| reduce_by_key{k,x,y->x+y}
| stdout
```

图 5-14 使用 Streem 进行单词计数

在图 5-14 中，下面的代码把从 stdin 输入的各行按照单词进行了分割。

```
flatmap{s->s.split(" ")}
```

split 函数根据作为参数传来的分隔符将字符串分割，然后返回分割后的数组。flatmap 把函数返回的数组展开为流元素。结果，当从上游的流传来以下 2 行数据后，

```
this is my pen
my name is yukihiro
```

流向下游的就是下面这个有 8 个元素的数据。

```
this
is
my
pen
my
name
is
yukihiro
```

另外，由于我想用图 5-14 来说明 flatmap，所以没有使用下面这个函数。

```
split(" ")
```

这是为了使代码更加简洁而准备的函数，它的作用与下面的 flatmap 代码基本相同。

```
flatmap{s->s.split(" ")}
```

下一个阶段的 map 会将单词序列转换为下面这种形式的数组。

```
[ 单词 , 1]
```

最后，reduce_by_key() 接收拥有 2 个元素的数组的流，按照每条数据中的第 1 个元素进行分组，对第 2 个元素进行归约。于是，原来的 2 行输入就会转换为下面这种形式。把这个结果输出到 stdout，就完成了单词计数的处理。

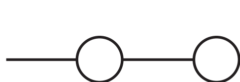
```
[this, 1]
[is, 2]
[my, 2]
[pen, 1]
[name, 1]
[yukihiro, 1]
```

流的分叉与合流

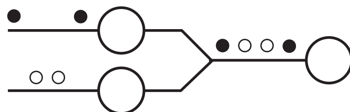
在前面介绍的大部分例子中，由流组成的管道是一条路走到底的模式，即从生产者开始，经过过滤器，最后到达消费者。

但是流的组成模式并不只有这一种（图 5-15）。

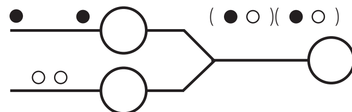
(1) 连接1–1



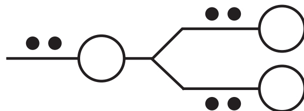
(2) 混合n–1



(3) 汇合（zip）n–1



(4) 分配1–n



(5) 结合2–1

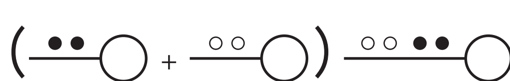


图 5-15　流的组成模式

图 5-15(1) 的连接指的是使用“|”将两个流的输入和输出相连，这就是前面出现的流的最基本的组成模式。

混合指的是把多个流的内容合并为一个流。数据按照到达的顺序混合后，流向下游的流。这里举一个具体的例子。比如有 a 和 b 两个流，当这两个流的内容流向 stdout 时，下面这两行代码就可以构建图 5-15(2) 所示的混合流。

```
a | stdout
b | stdout
```

图 5-15(3) 的汇合模式在合并多个流的数据这一点上与混合模式相同，但是行为不同。它并不是简单地让各个流的数据流向下游，而是对这些数据进行组合，形成数组之后再使其流向下游。

由于汇合模式看上去与拉锁（zipper）很像，所以就使用 zip 函数。

在用 seq() 生成从 1 开始的流，用 fread(path) 获得文件内容的流之后，假设为了把这两个流汇合为一个流而像下面这样使用了 zip 函数，

```
zip(seq(), fread(path))
```

这样一来，各行就都会被赋予如下所示的数据，其中每一行都是行号和行的内容的组合，正好相当于 cat-n 命令。

```
[数值, 行]
```

seq() 函数依次生成整数的速度与文件的读取速度应该是不同的。即便如此，各个流的元素也会按照顺序进行组合。

图 5-15(4) 的分配是指让从上游过来的数据流向多个流。让从 a 流过来的数据流向 b 和 c 这两个流的代码如下所示。

```
a | b
a | c
```

在这种情况下，流向 b 和 c 的是同样的数据。

最后的图 5-15(5) 的结合指的是把两个流按顺序连接起来。在结合 a 和 b 这两个流时，在 a 流的所有数据流向下游之后，再发送 b 流的数据。结合流时使用的运算符是“+”。比如运行下面这行代码，就会在 a 的内容之后输出 b 的内容。

```
(a + b) | stdout
```

UNIX 的 cat 命令会依次输出多个参数的文件内容，可以认为二者是同样的东西。

需要进行流量控制

这里出现了一个难题。比如在 zip 的情况下，输入的流有多个，每个流生成数据的速度不同。像 seq() 和 rand() 这种只进行简单计算的生产者会接连不断地发送数据，但是像套接字这样的生产者就会根据连接对象来决定发送数据的时机。

不过在没有集齐来自上游的所有数据之前，zip 本身是不会开始处理的。因此，在组合使用生成数据的速度不同的上游流时，速度快的流的数据就会被积压（消耗内存）。为了避免这样的情况，就需要进行数据的流量控制。

降低生产者的优先级

流量控制的实现方法有很多种，本次也将采取与前面相同的方式，通过降低生产者流的优先级来实现对流量的控制。

管道中流转的数据过多会使下游的流来不及处理，这时就需要进行流量控制，所以我打算让下游处理的优先级高于生产者，以此来进行流量控制。不过很难预测异步程序的举动。对于 zip 这种汇合两个流的函数，这种降低生产者优先级的方式能否奏效，现在还是未知数。所以要先让程序实际运行起来，之后再根据运行结果来确认是否有问题。

5-1 节之前的 Streem 实现都为运行队列设置了优先级，降低了生产者的优先级顺序。这次我准备了生产者专用的队列，通过把程序修改为按照普通队列、生产者队列的顺序取出数据，来降低生产者的优先级。

不考虑分配时的滞留

其实同样的问题也会在下游的流中发生。在数据被分配到多个下游的流时，如果各个下游的流的处理速度不同，那么在速度较慢的流之前就会出现数据的积压。但如果上游分配数据的流是过滤器，而不是生产者，那么前面介绍的通过降低生产者的优先级来进行流量控制的方法就没有效果了。

经过多番考虑，我认为下游的问题应该不会很严重，所以暂时不予以解决。

但是在这二十年来的 Ruby 开发的过程中，我明白了一点，那就是很难限制语言处理器的使用场景。可能会发生的问题必定会发生，甚至还会比预想的更加严重。将来有一天 Streem 也必须解决这个问题，我需要在这一天到来之前想好解决办法。

在目前使用的队列中增加对内存的考虑

为了实现流量控制，我决定大幅修改 Streem 事件处理的核心部分——事件循环。

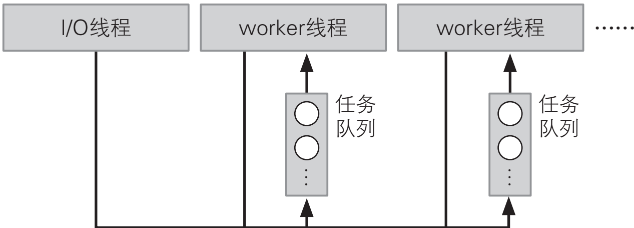


图 5-16 修改前的事件处理架构

图 5-16 所示为前面实现的事件处理架构。这个架构的特点是每个 worker 线程都有自己的任务队列，这样做的目的是让同一个管道的处理在同一个 worker 线程中执行。这样做有几个好处。首先，不用担心属于流的任务被多个线程执行，所以也就不需要考虑并发控制。另外，任务会被依次执行，所以不会出现处理速度不均导致数据顺序发生改变的情况。最后，由于一系列的任务在同一个线程执行，所以可以共享缓存，提高内存访问的效率。

但是在管道数量较少的情况下，即使在众核环境下运行，该架构也无法发挥出多核的威力，无法进一步提高性能。

也就是说，该架构虽然可以通过灵活使用缓存来提高性能，但也存在无法充分利用多核的缺点。综合考虑，缺点更明显一些。至于保证顺序这一点，由于在流处理的场景中顺序大多不会发展成很严重的问题，所以它也算不上是一个很大的优点。

修改为优先考虑内核的灵活使用

于是我设计了图 5-17 这种新的结构。共享队列之后，空闲的 worker 线程会去接收新的任务，从而提高内核使用率。属于同一个管道的任务如果在不同的线程执行，就可能无法共享一级缓存。不过数据保存在二级缓存和三级缓存的可能性非常高，所以这里忽略这个问题。另外，图 5-17 中省略了用于进行流量控制的生产者队列。

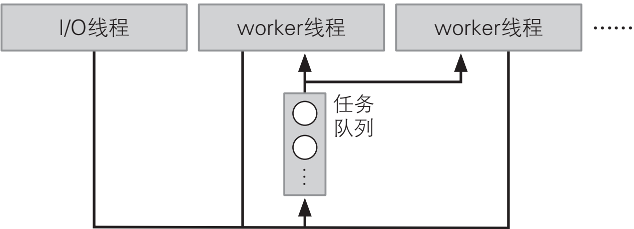


图 5-17 新的事件处理架构

不过，图 5-17 的架构还存在一些问题。首先，在属于同一个流的任务之间需要进行并发控制的情况非常多。在前面介绍的过滤器中，就有在流中存在状态的 `count()`、`sum()` 和 `reduce()` 等函数，这些函数就需要进行并发控制。

为了避免这种情况出现，我为每个 worker 线程都准备了属于自己的队列。如果从队列里取出的任务所属的流需要进行并发控制，就在当前任务运行结束之后，让接下来的任务继续运行，以避免因同时运行而失去连续性。换句话说，就是向正在运行任务的 worker 的队列里添加接下来的任务。

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

改善任务队列的实现

为了提高事件处理效率，我也修改了队列的实现。之前是通过组合使用锁和条件变量来实现队列的，这种实现方式虽然能够保证程序正常工作，但在性能上却有很大问题。

多线程环境下的数据结构如果不好好设计就不能正常工作。比如，在某个线程正在替换数据时，如果有别的线程读取数据，那么实际读到的数据就可能是下面几种情况的其中一种。

- 运气很好，是正确的数据
- 是数据替换过程中的不完整的数据
- 是已经替换完毕的毫无关系的垃圾数据

更糟的是，不同的运行时机会发生不同的情况，你根本无法预测到底会发生哪种情况。基本上可以正常工作，但偶尔会失败，这种 bug 才是最难解决的。

为了避免这种问题出现，需要在多线程环境下对共享的数据结构进行并发控制。具体来说，在访问线程之间共享的数据时，要在访问之前加锁，访问之后解锁。虽然其他线程在访问数据时也会去加锁，但如果由于有线程正在访问而已经上了锁，那么准备加锁的线程就需要在锁被释放之前暂停运行。

暂停运行就意味着处理停滞。Streem 会启动与内核数数量相同的 worker 线程，对于这种类型的架构来说，这就意味着无法充分利用宝贵的多核资源，真的很浪费。

为了避免这样的问题出现，就需要设计出不使用锁进行保护的无锁数据结构。无锁数据结构采用了无论多个线程在何时访问，数据都不会出现问题的算法。

这些全部都由我自己来实现有些不太现实，所以我决定利用现成的东西。我在 GitHub 上找到了下面这个库，我决定（在进行大幅改造之后）使用这个无锁队列。

```
github:supermartian/lockfree-queue
```

CAS

在不考虑线程的普通编程中，更新数据的操作步骤是先读取，然后再加工、写回。但是，在读取之后加工之前的这段时间里，如果有其他线程访问或者修改了这个数据，可能就会出现数据不完整的问题。为了不发生这样的问题，就需要让数据的查看和修改同时进行，这就是无锁算法的基本思想。

这种不可分割的修改操作被称为 CAS（Compare and Swap，比较并交换）。CAS 有内存位置、值 1 和值 2 这 3 个参数。比较内存位置的值与值 1，如果相等就把值 2 保存到内存位置。CAS 是通过 CPU 的 1 个指令实现的，可以保证在操作过程中不会有其他线程执行。这种不可分割的操作就称为原子操作。

C 语言最开始不存在实现 CAS 的指令，所以需要用汇编语言编写调用 CAS 指令的代码。不过幸运的是，GCC 从 4.1 版本开始提供了实现 CAS 的扩展功能，该功能的代码如下所示。

```
__sync_bool_compare_and_swap()
```

小结

本节对管道处理的概念进行了整理，并对实现进行了改进。在多线程环境中编程时需要考虑很多事情，很让人烦恼。下一期（本书中是 4-5 节，具体请查看下面的时光机专栏）我想再稍微介绍一下这个实现的改进。

时光机专栏

依赖于运行时机的bug真是难以处理

本节是 2016 年 4 月刊中刊登的内容。本节新增了 `reduce`、`reduce_by_key` 和 `zip` 等函数作为管道的构成要素。另外，我们也初步接触了无锁队列。

连载时本节内容刊登在 4-5 节的前一期，所以可能有人会觉得本书的顺序有点别扭。在 4-5 节的时光机专栏里我提到过，无锁队列并没有解决高负荷时发生的 bug，所以最终没有采用（虽然代码还在，但是编译时还是通过标志位使用了有锁队列）。

依赖于运行时机的 bug 真的很难调试。我花了很大精力去尝试解决这个 bug，但是因时间有限，最后只好选择放弃，以后我会再去挑战这个问题的。

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

5-3 CSV处理功能

当进行像Streem这样的管道处理时，作为输入的数据格式，最常见的恐怕就是CSV格式了。本节将一边开发CSV格式的数据输入处理功能，一边进行相关内容的讲解。非“标准”的CSV处理相当麻烦。

CSV（Comma-Separated Values，逗号分隔值）作为表示表格结构的数据格式被广泛使用。特别是在把 Excel 等计算软件的数据导出到其他软件时，CSV 这种简单的文本格式是最靠谱的，也是最让人放心的。

因为各大软件都支持 CSV 格式，所以它没有一个统一的标准。尽管 IETF（Internet Engineering Task Force，国际互联网工程任务组）以文件的形式定义了 RFC4180 规范，但这也只是作为信息参考使用，并不是一个严密的规范，而且很多 CSV 数据没有遵循 RFC4180 规范。不过尽管如此，我们也不能忽视该规范。

本节我们就来为 Streem 添加 CSV 格式的数据输入处理功能。

RFC4180

RFC4180 定义的 CSV 数据格式的规则大体上可以总结为如下内容。

首先，文件需包含一条以上的记录。记录是由 CRLF（Carriage Return/Line Feed）分隔的“行”，用 C 语言的风格表示 CRLF 就是 `"\r\n"`。

记录需包含一个以上的字段。字段由逗号分隔，最后一个字段的后面没有逗号。RFC4180 规定每条记录都包含同样数量的字段。

字段可以用双引号“`"`”围起来，但是如果字段中包含逗号、双引号和换行等字符，就必须用双引号围起来。在被双引号围起来的字段中使用接连出现的两个双引号“`""`”来表示双引号本身。

CSV 可以带有表头（header）。是否带有表头由外部指定，如果有表头，那么构成表头这条记录的各字段的字符串就相当于各字段的名称。

CSV 的变体

前面提到过，RFC4180 不是 CSV 规范，更像是一个松散的协议。在 RFC4180 被制定之前就已经开发出了很多解释 CSV 的软件，所以 CSV 的解释也有很多变体，比如下面这些。

- 记录的分隔符不仅有 CRLF，有的软件也允许用 LF
- 字段的分隔符不仅有逗号，有的软件也允许用 tab 和空格
- 双引号的转义是把双引号重复两次，不过有的软件也允许以前置反斜杠“\”的方式来进行转义
- RFC4180 要求所有记录的字段数都相同。在字段数不同的情况下，各软件的处理方式也不尽相同，有的软件报错，有的软件则忽略多余的字段，在末尾用空字符补上不足的字段
- 在 CSV 文件中有空行的情况下，有的软件会忽略这一行，有的软件则会把这一行作为字段数为 0 的记录处理
- 有的软件允许 CSV 数据中存在注释
- RFC4180 中没有关于字段的数据类型的描述（全部作为字符串处理），有的软件会自动把全部由数字构成的字段转换为数值

由于 CSV 格式的数据非常多，所以我们无法忽视这些变体的存在，许多 CSV 函数都接受大量的可选形式。

不过这次我不考虑这些可选形式，而是以在大多数情况下可以正常运行为目标，今后再根据需要强化功能。

探索 GitHub

虽然我也可以自己从零开始编写解释 CSV 的函数，但既然自己开发的是开源软件，就去借鉴一下他人的成果吧。

于是我决定在网上查找解释 CSV 的开源函数，从中选择符合条件的来使用，如果找不到就自己去开发。

CSV 函数的查找条件如下所示。

- 代码是用 C 语言开发的，这样容易引入到 Streem 中
- 代码要容易理解，以便后面修改
- 要保证是线程安全的，比如没有使用全局变量等
- 许可证要适合与 Streem 组合使用，最好是 MIT

那么有满足这些条件的库吗？总之先去 GitHub 上找找看。现在源代码都集中在 GitHub 里，找起来轻松多了。

在 GitHub 的搜索栏里，输入下面的查询条件。

```
csv language:C
```

language:C 部分用来指定开发语言。这一信息不是用户指定的，而是 GitHub 根据源代码的信息推测出来的，所有偶尔会出现错误。不过出错的基本上是用多种语言实现的工程，这次要找的功能单一的库应该不会出错。

用这个条件找到了 174 个代码库（2015 年 7 月上旬的结果），数量不少。先排除掉其中没有明示许可证的库。如果剩下的库中没有符合条件的，就需要向作者留言进行交涉了，所以还是一开始就写清楚许可证的库省事一些。

接着基于上面的条件对各个库进行筛选。符合条件的库中数量最多的是以 GPL 协议提供的 libcvs 的封装库，但由于这次用不到 libcvs 这么多的功能，所以排除掉这些库。其次比较多的是全局变量的库，由于这些库不适合在多线程运行的 Streem 中使用，所以也只好放弃。

经过一番筛选之后，剩下的库中最符合条件的就是名为 semitrivial/csv_parser 的库了。该库没有使用全局变量，非常简单，也没有多余的处理，而且代码容易理解，改造起来应该也很容易。更难得的是，许可证也与 Streem 本身一致，都是 MIT 许可证。因此，我决定以这个库为基础进行开发。

许可证

GitHub 里很多库都没有写明许可证。所幸这次我找到了 MIT 许可证的源代码，但是在没有写明许可证的情况下，我们该怎么办呢？

推荐的做法是尝试直接联系作者。既可以用 GitHub 的 issue 进行提问，也可以给作者发邮件，因为大多数情况下作者会留下自己的邮箱地址。比如我们可以在邮件中这样写："我想在自己的软件里使用您的源代码，可是源代码没有写明许可证，我用起来不放心，所以想请您确定一下许可证。我个人希望是 MIT 许可证。如果可以，我会以 ×× 方式感谢您，您意下如何？"如果作者不是过于以自我为中心，我想应该会积极地回应。

在 GitHub 公开源代码的人，一般不会反感对自己的代码感兴趣的人来联系自己。但如果要求对方按照自己的需求修改已经拥有一定用户的软件的许可证，恐怕对方不会同意。大家换位思考一下就能明白。在这一点上，比起已经声明了许可证的软件，请求没有声明许可证的软件的作者为我们加上许可证或许更加可行（虽然有得不到回应的风险）。

其实这次采用的 semitrivial/csv_parser 库在我写稿期间被原作者删除了。虽然我手头有副本，不影响工作，但如果不得到原作者的同意，说不定以后会引起纠纷。

于是我尝试联系了作者，并很快得到了作者的回信，信中说："我没想到是这么重要的代码，所以没有多想就删除了。你要用的话我就恢复这个库。谢谢你喜欢这个代码。"后来为了对作者恢复代码表示感谢，我 fork 了这个库，并加了星。

这样的交流也是开源软件开发的乐趣之一。

csv_parser

原版的 csv_parser 由图 5-18 的两个函数组成。parse_csv 函数解析字符串形式的 CSV 记录，返回分割成字段的字符串数组。使用完这个数组后，用 free_csv_line 函数释放内存。

```
char **parse_csv(const char *line);
void free_csv_line(char **parsed);
```

图 5-18　csv_parser 提供的函数

包含空行在内，整个实现只有 145 行。可以说这个代码非常容易理解，改造起来也很容易。

不过在添加到 Streem 的代码中时，有几处需要进行修改。首先是字符串的表示。csv_parser 接收末尾是 NUL（\0）的 C 字符串，并返回 C 字符串的数组。但由于 Streem 有自己的对象，所以需要接收 Streem 字符串（strm_string），并返回包含 Streem 字符串的 Streem 数组（strm_array）。

先从这里开始修改，另外还要支持中间包含 NUL 的字符串。

首先，将接收末尾为 NUL 的 C 字符串数据的部分全部替换为 Streem 字符串（strm_string*）。另外，把循环条件“不包括 NUL 的字符串长度”修改为“字符串的长度”。

在返回数据部分的代码中，将由 malloc 分配内存空间的部分替换为 Streem 对象的分配（数组由 strm_ary_new 分配，字符串由 strm_str_value 分配）。

原版不支持末尾换行，所以当末尾有换行符（CR 或者 LF）时，要将其删除。

内存管理由 Streem 的 GC 功能负责。删除图 5-18 的 free_csv_line 函数。

Streem 数组

这样就（姑且）在 Streem 中添加了 CSV 功能，图 5-19 的代码可以运行了。

```
fread("sample.csv")|csv()|stdout
```

图 5-19　读取 CSV 的 Streem 代码

不过现在 Streem 还没有输出数组的功能，所以即使读取 CSV 文件并输出到 stdout，转换为数组的记录也只能表示为如下形式。

```
[...]
```

这样一来，即便知道传来的是数组，因为不了解信息的内容，所以就连代码是否正常工作也都不得而知，这都是拜我之前偷懒所赐。

这样下去是不行的，所以借此机会我要实现表示数组的功能。

数组的表示其实非常麻烦。数组是递归的数据结构，包括数组本身在内的其他数据也都可以作为数组的元素。另外，在字符串的情况下，表示方法又会根据字符串是否为数组元素而发生改变。

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

具体来说，比如在输出 "abc\n" 这个字符串时，会按照 a、b、c 以及换行的顺序输出。如果这个字符串是数组元素，输出就应该是如下这种形式。

```
["abc\n"]
```

也就是说，作为数组元素的字符串要用双引号围起来，特殊字符要转换为转义形式。

大家可以参考 value.c 文件里的 strm_inspect 函数，处理大体上会按照下列步骤进行。

- 除了现有的字符串输出函数（strm_to_str 函数）之外，还新增了更容易理解的字符串输出函数（strm_inspect 函数）
- 使用 strm_inspect 函数进行数组的字符串输出。strm_inspect 函数对各种数据类型进行如下的字符串输出处理
  - 对于整数和浮点数，直接进行字符串输出。字符串用双引号围住，特殊字符（\t、\r、\n、\e）输出为转义形式，控制字符输出为数值
  - 对于数组，在“[”之后用 strm_inspect 函数依次对各个元素进行字符串输出，字符串输出后的各元素用“,”隔开，在末尾输出“]”

这样就可以用人能够理解的形式输出数组了，比如读入下面这行代码后，

```
松本,男性,50\r\n
```

就会输出以下内容。

```
["松本","男性","50"]
```

CSV 格式

虽然可以读取 CSV 数据了，但是我们做得还不够完美。

针对前面提到的 CSV 格式不统一的问题，我们需要去探讨 Streem 该如何处理，并认真地做出决定。

这里再整理一下 CSV 格式不统一的地方，主要表现在以下几点。

- 记录的分隔符
- 字段的分割符
- 字段数量
- 引号转义

- 注释
- 字段类型
- 表头

对于这些不统一的地方，Streem 不认可变体形式。Streem 实际支持的情况如表 5-9 所示。比起通过支持各种变体来应对各种各样的情况（这么做可能会让代码变得复杂），我更看重代码的简洁程度。

表 5-9 Streem 支持的 CSV 功能

功能	不统一的地方	Streem 支持情况
记录的分隔符	LF、CRLF	LF（删除末尾的 CR）
字段的分割符	逗号或者 tab	只支持逗号
字段数量	报错或者调整至相同的字段数量	报错
引号转义	两个引号、前置反斜杠“\”	两个引号
注释	#、// 等	无
字段类型	全部是字符串、数值或者日期	只会自动转换数值
表头	从外部指定	如果第一行全部是字符串，则视为表头

将来也许需要支持一部分变体形式，不过目前我们先这样进行开发。

CSV 的任务化

接下来就开始对原版的 csv_parser 进行改造。

首先是事前准备，让 csv() 函数返回专用的任务。实现方法类似于 4-1 节介绍的服务器端套接字的实现。

先创建用于保存在任务之间共享的数据的结构体，把这个结构体命名为 csv_data（图 5-20）。我会在后面介绍结构体成员的含义。

csv 函数的代码如图 5-21 所示。该函数所做的处理只是初始化 csv_data 结构体，创建任务而已。

```
struct csv_data {
  strm_array *headers;
  enum csv_type *types;
  strm_string *prev;
  int n;
};
```

图 5-20 csv_data 结构体

```
static int
csv(strm_state* state, int argc, strm_value* args, strm_value* ret) {
  strm_task *task;
  struct csv_data *cd = malloc(sizeof(struct csv_data));

  if (!cd) return STRM_NG;
  cd->headers = NULL;
  cd->types = NULL;
  cd->prev = NULL;
  cd->n = 0;

  task = strm_task_new(strm_filter, csv_accept, NULL, (void*)cd);
  *ret = strm_task_value(task);
  return STRM_OK;
}
```

图 5-21 csv 函数的代码

这样一来，当数据从管道的上游传过来时，`csv_accept` 函数就会运行。这与 4-1 节介绍的 `tcp_server` 函数完全相同。`csv_accept` 函数的定义如图 5-22 所示。从上游传来的数据是第 2 个参数。另外，刚才被初始化的 `csv_data` 结构体可以通过 `task->data` 的形式获取。不过因为它是以“`void*`”的形式保存的，所以需要通过类型转换将其恢复为原有结构体的指针。

```
static void
csv_accept(strm_task* task, strm_value data) {
  strm_string *line = strm_value_str(data);
  struct csv_data *cd = task->data;
  ...
}
```

图 5-22 csv_accept 函数的定义

检查字段数

走到这一步之后，剩下的就是依次实现表 5-9 里的功能了。

首先从检查字段数开始实现。如果 CSV 各条记录的字段数不同，就报错。以前讲过，Streem 的管道处理出现错误时会忽略错误的数据，所以当不满足条件时，程序只会在那里 `return` 而已。例如在检查字段数时添加图 5-23 的代码。

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

```
if (cd->n > 0 && fieldcnt != cd->n)
  return;
cd->n = fieldcnt;
```

图 5-23 csv_accept 函数的字段数检查

由于 cd->n 会被初始化为 0，所以它在 csv_accept 首次运行之前的值是 0。在首次运行时，cd->n 会被设置为本次的字段数，然后用于下一次检查。这就意味着所有与第一条记录的字段数不同的记录都会被忽略。

仔细想想，这个为了便于开发而确定的忽略错误数据的策略在实用层面上并没有什么问题，但是在开发程序时如果发生了 bug，就很难找出问题发生在哪里，所以我认为至少在开发模式下要能收到错误消息。近期我准备抽出一些时间来修改错误处理。

多行记录

CSV 允许字符串中存在具有特殊含义的字符，比如逗号、（转义过的）双引号和换行。

不过在目前的 Streem 的 CSV 处理中，读取文件时从上游拿到的是已经被分割为行的数据，因此当字符串中包含换行时，本应作为一条记录的数据就会被分割为多行，然后再进行传递。

于是这里添加了图 5-24 的代码，如果字符串在引号中结束了（也就是输入的数据是包含换行的字符串），就把它保存在 csv_data 结构体里，在下一次运行时把它与接下来传过来的数据进行结合，并解释为一条记录。这利用了字符串在引号中结束后计算字段数的 count_fields 函数会返回 -1 这一性质。

```
if (cd->prev) {
  strm_string *str = strm_str_new(NULL, cd->prev->len+line->len+1);

  tmp = (char*)str->ptr;
  memcpy(tmp, cd->prev->ptr, cd->prev->len);
  *(tmp+cd->prev->len) = '\n';
  memcpy(tmp+cd->prev->len+1, line->ptr, line->len);
  line = str;
  cd->prev = NULL;
}
fieldcnt = count_fields(line);
if (fieldcnt == -1) {
  cd->prev = line;
  return;
}
```

图 5-24 多行处理

如果前面有行（cd->prev）存在，就需要先把前面的行和现在的行结合起来。目前结合处理部分的代码还不够美观，我想在以后去改进它。

字段类型

目前所有字段都是作为字符串处理的，但对于明显是表示数值的字段，大多数情况下还是将其自动转换为数值比较方便。因此，当组成字段的字符串全部是数字时，就将其自动转换为整数，当两个数字序列之间用“.”隔开时，就将其自动转换为浮点数。这是通过把处理字符串的部分抽出为 csv_value 函数，并在其中增加根据内容转换为整数或浮点数的处理来实现的。

另外，我还决定把类型信息记录在 csv_data 的 types 成员里，在字段类型不匹配的情况下报错，跳过对那条记录的处理。

表头处理

很多 CSV 数据在第一行都带有表头。接下来我们就来添加表头处理的功能。

R 语言的 data.table 库中有一个读取 CSV 数据的 fread 函数，当文件的第一行全部是字符串时，这个函数就会把它当作表头。我们可以参考这种做法，在同样的条件成立时，把第一行视为表头，并跳过这一行。

不过，跳过这一行会丢失一些信息，这让我觉得很可惜。

所以我决定在带表头的 CSV 数据的解析结果中返回持有表头指定的字段名的 map。

本来 Streem 的语法中就有与 Ruby 等的散列类似的 map 数据类型，只是 Streem 的数据具有不可变的特点，所以我觉得散列没有什么实际意义，虽然在语法上进行了支持，但是并没有去实现。

不过我打算趁这次机会把 map 重新定义为“元素带有名称的数组”。这并不是什么罕见的数据类型，Python 就以“带名称的元组”的形式引入了这个数据结构，R 里面也有类似的功能。

于是我在表示数组的结构体 strm_array* 里增加了 headers 成员，当这个结构体中包含名称信息时，就用 strm_inspect 函数显示每个元素的名称信息。虽然我也想使用名称来进行访问，但是由于时间的关系，只好以后再实现这个功能了。

最终，带表头的 CSV 文件会像图 5-25 那样被解析。

```
■输入
名字,出生地
matz,鸟取县
junko,山口县

■输出
[名字:"matz",出生地:"鸟取县"]
[名字:"junko",出生地:"山口县"]
```

图 5-25　CSV 的表头处理

错误运行

我添加了支持类型和表头的功能，想让 CSV 的解析功能变得更智能一些，可实际用 CSV 数据

进行测试时，却出现了几种我不愿看到的情况。

首先是类型不一致导致数据被跳过的情况。用于测试 CSV 功能的数据中，各字段类型不一致的情况相当多，这些类型不一致的记录都会被自动跳过。虽说符合要求，但如果忘记了这一点，就会非常纳闷，不明白为何会出现这种状况。不过实际常用的 CSV 数据大部分应该是类型一致的，所以应该没什么大问题。

另一个是表头。其实有不少 CSV 数据是没有表头的，而且所有字段都是字符串。目前的实现会把正常的记录当成表头，这样就丢失了第一行的数据。这是非常严重的问题，所以要么允许用户通过可选项指定是否有表头，要么设计出更加智能的表头判断条件，否则该功能可能就无法达到实用程度。

通过这次实现我认识到 CSV 的解析非常复杂，现在还不能说已经实现了这个功能。尤其是上述两个问题非常严重，还需要进行修改。

小结

如果不对照着源代码来阅读本节内容，恐怕不容易理解。大家可以下载 https://github.com/matz/streem.git 里的源代码，一边看 src/csv.c 一边阅读本节，将有助于理解。本节内容对应的源代码的标签是 201509。

时光机专栏

CSV处理是很久以前实现的

本节是 2015 年 9 月刊中刊登的内容。Streem 的基本部分已经差不多完成了，后面就以添加功能（以及提高完成度和改善性能）为主。首先着手实现的就是对数据处理语言来说非常基本的 CSV 读取功能。

也许读者已经注意到了，本节比前后几节的连载日期都要早。虽然在成书时把这一节内容放在了后面，但在连载中它很早就出现了。本节提到的把数组与 map 看作同样的数据类型这一点，其实是 3-2 节介绍的 Streem 面向对象功能的基础。如果有时间重读本书，按照杂志连载顺序来读也别有一番趣味。

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

5-4 时间表示

本节将介绍时间表示和时间操作的实现：按照国际标准表示时间，同时提供时区支持，以及进行时间的加法和减法运算。由于UNIX标准的API不够丰富，所以我在设计和实现时花费了很多精力。

本节我们来思考一下时间表示和时间操作的实现。在流数据处理中，有时也需要进行时间操作。

我们以一个任务为例来进行说明。假设要根据东京的降水量数据绘制图表，也就是说，要读取某年某月某日下了几毫米的雨这种 CSV 数据，然后绘制成图表，也许还要计算月平均降水量或者年平均降水量等。在这种情况下，就需要把时间作为数据进行处理了。

时间（或者时刻）在多数情况下使用以下字符串表示。

```
2016-06-01
```

字符串虽然可以比较大小，但是不容易计算经过的天数等。

在编程中，把时间作为时间类型来处理，是最正常不过的需求。

时间与时刻

“时间”这个单词有多种含义，既可以用来表示时间的长度（例如撰写这篇稿件花了 2 周的时间），也可以用来表示某个瞬间的时间的位置（例如现在的时间是 2016 年 5 月 1 日上午 10 点）。这种模棱两可的表达经常会招致混乱，所以后面我会用“时刻”来表示某个特定的时间的位置，用英语表示就是“time stamp”。

而时间的长度用英语表示是“duration”，目前我还没有想到合适的词语，应该可以称之为“时间间隔”“持续时间”等吧。也有人特意把“时间”这个词限定为这个含义来使用，但我觉得有时这也会让人混乱。

用字符串表示时刻

时刻的表示方法各种各样，尤其是日期的表示方法与文化有着紧密关系，比如日本人会使用下面这种形式表示日期。

```
平成 28 年 5 月 1 日
```

而美国人则经常使用下面这种形式。

```
May 1 2016
```

到了欧洲，日期的表示顺序就变成了下面这样。

```
1 May 2016
```

日期的表示方法不统一会导致混乱，于是规定了日期和时刻的表示方法的国际标准便应运而生，这个标准就是 ISO 8601。

ISO 8601 使用下面这两种形式表示日期。

```
20160501（基本形式）
```

或者

```
2016-05-01（扩展形式）
```

这两种形式都是按照年→月→日的顺序来表示日期的。如果日期中也包含经过的时间（duration），就写成

```
20160501T100000+0900
```

或者

```
2016-05-01T10:00:00+09:00
```

由于基本形式难以与数值区分，所以大多数情况下会使用扩展形式来表示日期。

时刻的表示方法

使用 ISO 8601 就可以用字符串表示时刻了，但是用字符串表示时刻不便于进行程序处理，因此我考虑引入时刻类型。那么，如何表示时刻才比较合适呢？

时间是从过去到未来一维地流逝的东西，因此用数值表示比较合适。很多系统将某个特定的时刻（称为纪元时刻）作为原点，用从这个原点开始经过的时间来表示时刻。

包含 Linux 在内的很多 UNIX 系统把 1970 年 1 月 1 日 00:00（UTC）作为纪元时刻，并用秒表示经过的时间，所以 2016-05-01T10:00Z 可以用从纪元时刻 1970-01-01T00:00 开始经过的秒数 1 462 096 800 来表示。例如，获取当前时刻的系统调用 `time(2)` 是下面这个 API。

```
time_t t = time(NULL);
```

`t` 是以整数形式返回的从纪元时刻开始经过的秒数。

不过并不是所有问题都能用秒单位解决，有时也需要取出 1 秒以下的信息。UNIX 可能是考虑到了这一点，从而增加了新的系统调用 `gettimeofday(2)`，这个调用把秒以下的时刻用微秒这个单位返回。

```
struct timeval tv;
gettimeofday(&tv, NULL);
tv.tv_sec; // => 秒数
tv.tv_usec; // => 微秒
```

大家可能会感到奇怪："明明是微秒，为什么变量名用 `usec` 呢?" 据说这是因为字母 u 与表示"微"（一百万分之一）的希腊字母 μ（miu）形似。

后来，可能因为需要更加精确地分解时刻，POSIX.1-2008 中增加了系统调用 `clock_gettime(2)`。

```
struct timespec tp;
clock_gettime(CLOCK_REALTIME,&tp)
tp.tv_sec; // => 秒数
tp.tv_nsec; // => 纳秒
```

`clock_gettime(2)` 以纳秒为单位来表示不足 1 秒的部分。

虽然 UNIX 使用从 1970 年 1 月 1 日开始的秒数来表示时刻，但并不是所有系统都是这么做的。比如 Windows NT 就把 1601 年 1 月 1 日当作纪元时刻，并以 100 纳秒为单位来表示经过的时间。

时刻类型的结构体

下面我们就把表示时刻的数据类型添加到 Streem 里，实现方法与以前介绍的 `kvs` 的实现方法基本相同。

首先定义表示时刻的结构体（图 5-26）。在拥有方法的结构体之前配置下面这个宏。

```
struct strm_time {
  STRM_AUX_HEADER;
  struct timeval tv;
  int utc_offset;
};
```

图 5-26　时刻类型的结构体

```
STRM_AUX_HEADER;
```

然后定义表示实际时刻的类型 struct timeval 和表示时差的整数 utc_offset。struct timeval 结构体的定义如图 5-27 所示，它以微秒级的精度来表示时刻。

```
struct timeval {
  time_t      tv_sec;      /* seconds */
  suseconds_t tv_usec;     /* microseconds */
};
```

图 5-27 struct timeval 的定义

在决定使用 struct timeval 时，我注意到 man 文档里有图 5-28 那样的描述。“不推荐”是说状态不稳定。obsolete 表示“已过时”，与“将被废弃”的意思相近。

```
POSIX.1-2008 marks gettimeofday() as
obsolete, recommending the use of clock_
gettime(2) instead.

翻译：POSIX.1-2008不推荐使用gettimeofday()，推荐
使用clock_gettime(2)。
```

图 5-28 gettimeofday 的 man 文档的记载

就在我想不得不去认真考虑 clock_gettime() 的使用时，调查中发现 Mac OS 好像到现在都还没有实现 clock_gettime()。虽然很难想象 Mac OS 到现在还没有实现 POSIX.1-2008 这个已经定义了很多年的标准，但是考虑到兼容性的问题，我只好放弃使用 clock_gettime()，最后决定使用可靠且口碑良好的 gettimeofday()。

utc_offset 以秒为单位表示与 UTC 的时差，比如日本比 UTC 早了 9 个小时，那么表示日本时刻的 utc_offset 就是下面这个值。

```
9 × 60 × 60 = 32400
```

从原理的角度考虑，其实时刻类型中不需要时差信息。对于表示某个瞬间的时刻来说，时差是没有意义的。日本的晚上 9 点，也就是 UTC 的正午，在本质上是同一个时刻，只不过表示方式不同而已。

把时刻转换为字符串形式时需要使用时差信息。“某个时刻是 9 点”这一信息，在没有时区信息的情况下是没有意义的。

对于有时区信息的时刻，比如以下面这种方式表示的时刻，一般情况下都希望默认以本身的时区信息表示。

```
2016-05-01T10:00:00+09:00
```

所以我决定在指定了时区的时刻数据里默认插入时差信息。

UTC

在前面的内容中 UTC 这个术语出现了多次，我都还没有向大家介绍，下面就来详细介绍一下。UTC（Coordinated Universal Time，协调世界时）是时刻的原点。至于为什么没有按照首字母取名为 CUT，其中有着比较复杂的原因。据我了解，在制定规范时，关于这个用语的正式名称，在英语 Coordinated Universal Time 和法语 Temps Universel Coordonne 之间发生了激烈的争执，最终作为妥协采用了两者之外的 UTC。

在这之前，时刻的原点被称为 GMT（Greenwich Mean Time，格林尼治平时），这个名字源自英国格林尼治天文台。与通过观测天体得出的 GMT 时间相比，UTC 追求更加精确的时间，它是通过将铯 -133 振动 91 亿 9263 万 1770 次的时间作为 1 秒的原子钟求得的。

由于地球的自转速度不固定，所以 GMT 和 UTC 之间会产生微小的差异。为了修正这个差异，需要在极少数的情况下插入闰秒。

闰秒是一个很麻烦的问题，比如在 2012 年 7 月 1 日插入闰秒后就引起了大范围的问题。

时刻类型数据的生成

下面就来思考一下如何生成时刻类型的数据。生成当前时刻的 `now()` 函数的实现如图 5-29 所示。`now()` 有一个可以省略的参数，如果指定了这个参数，其值就会成为时区信息（表 5-10）。

```
static int
time_now(strm_stream* strm, int argc, strm_value* args, strm_value* ret)
{
  struct timeval tv;
  int utc_offset;

  switch (argc) {
  case 0:
    utc_offset = time_localoffset();
    break;
  case 1:                       /* timezone */
    {
      strm_string str = strm_value_str(args[0]);
      utc_offset = parse_tz(strm_str_ptr(str), strm_str_len(str));
      if (utc_offset == TZ_FAIL) {
        strm_raise(strm, "wrong timezeone");
        return STRM_NG;
      }
    }
```

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

```
    break;
  default:
    strm_raise(strm, "wrong # of arguments");
    return STRM_NG;
  }
  gettimeofday(&tv, NULL);
  return time_alloc(&tv, utc_offset, ret);
}
```

图 5-29　now() 的实现

还有其他指定时区的方法，比如使用“JST”这样的省略形式表示日本标准时间，或者使用“Asia/Tokyo”这样的城市名来指定时区。但是 Japan Standard Time 与 Jamaica Standard Time（假设存在这个标准）可能会发生混淆，而且还需要世界各地城市名的列表，所以这次我没有采用这些方法。

表 5-10　指定时区的写法

写法	含义
Z	UTC
+09:00	UTC+9 时间
+0900	UTC+9 时间（省略形式）
+900	UTC+9 时间（进一步省略的形式）
-09:00	UTC–9 时间
-0900	UTC–9 时间（省略形式）
-900	UTC–9 时间（进一步省略的形式）

`time_now()` 只负责处理参数，通过 `gettimeofday(2)` 获取当前时刻，调用 `time_alloc()`。

如图 5-30 所示，`time_alloc()` 的实现一点都不难。在进行了 Streem 对象初始化（`type` 和 `ns` 的设置）之后，对传来的 `struct timeval` 类型的 `tv_usec` 进行异常值检查，如果是负值或者超过 1 秒的值，就对其进行调整。

```
static int
time_alloc(struct timeval* tv, int utc_offset, strm_value* ret)
{
  struct strm_time* t = malloc(sizeof(struct strm_time));

  if (!t) return STRM_NG;
  t->type = STRM_PTR_AUX;
  t->ns = time_ns;
  while (tv->tv_usec < 0) {
    tv->tv_sec--;
    tv->tv_usec += 1000000;
  }
  while (tv->tv_usec >= 1000000) {
    tv->tv_sec++;
    tv->tv_usec -= 1000000;
  }
```

```
  memcpy(&t->tv, tv, sizeof(struct timeval));
  t->utc_offset = utc_offset;
  *ret = strm_ptr_value(t);
  return STRM_OK;
}
```

图 5-30 time_alloc() 的实现

时差的计算方法

实际上，比较难实现的不是上面讲的这些，而是计算本地时间与 UTC 的时差的 `time_localoffset()` 函数。

我一开始想到的办法是通过计算本地时间与 1970-01-01T00:00:00 相对应的时刻来求时差。不过我通过 Twitter 了解到在某些情况下 1970 年与现在的时差存在不同。

确实，由于日本的时区基本没有变化，所以大家很容易忘记很多国家引入了夏令时。据说日本也在 1948 年到 1952 年的这段时间实施了夏令时。这样看来，在很多情况下不能以 1970 年 1 月为基准来计算时差。

正当我苦恼该如何去做的时候，从 Twitter 上知道了使用 `gmtime(3)` 和 `mktime(3)` 可以轻松求得时差[1]，而且代码还不到 140 个字符，真是令人惊讶。

我以在 Twitter 上学到的代码为基础，并对其进行了一些改造，形成了如图 5-31 所示的代码。严格来说，图 5-31 的代码还不支持在程序运行过程中时区发生变化的情况，这里我们先不考虑这个问题。如果将来在世界到处飞的人的设备上也运行 Streem 了，那时再考虑这个问题。

虽然我是这么想的，但由于有些国家要进行冬令时和夏令时的切换，而切换时程序很可能正在运行，所以还是要考虑如何解决这个问题。

这段代码的关键在于 `gmtime(3)` 的使用

```
static int
time_localoffset()
{
  static int localoffset = 1;

  if (localoffset == 1) {
    time_t now;
    struct tm gm;
    double d;

    now = time(NULL);
    gmtime_r(&now, &gm);
    d = difftime(now, mktime(&gm));
    localoffset = d;
  }
  return localoffset;
}
```

图 5-31 本地时间的时差的计算方法

① https://twitter.com/unak/status/717026294337122304

方法。gmtime(3)函数把 time_t 表示的以秒为单位的当前时刻，转换为分开表示 UTC 的日期和时间的 struct tm。转换为本地时间的 struct tm 的函数是 localtime(3)（带 _r 的是它的线程安全版）。另外，mktime(3)函数执行与 localtime(3)相反的操作，将 struct tm 转换为 time_t。

把使用 gmtime(3)得到的 UTC 形式的日期用 mktime(3)转换为本地时间形式的 time_t，就可以得到只有时差部分不同的时刻。之后再用 difftime(3)计算时间间隔，就可以得到以秒为单位的时差了。

UNIX 的时间函数有以秒为单位的，也有以微秒为单位的，还有以纳秒为单位的，这些时间函数都混杂在一起，而且根据平台的不同，有的函数能用，有的不能用，另外处理时差的函数也不够全面，所以老实说不是很好用，而我试着把各个功能组合起来之后，没想到效果还不错。

时刻操作的实现

下面来定义时刻类型的方法，我们暂时先定义表 5-11 中的 4 个方法。

表 5-11　时刻类型的方法

名称	功能	备考
+	时刻的加法运算	时刻 + 数值（秒数）→时刻
–	时刻的减法运算	“时刻 – 时刻”或者“时刻 – 数值”
number	转换为浮点数	用浮点数获取从纪元时刻开始的秒数
string	转换为字符串	时区参数可以省略

需要注意的是减法运算方法的类型。从时刻减去时刻可以得到经过时间，这个结果该用什么类型表示是一个令人头疼的问题。可以想到的方案有引入表示 duration 的数据类型，以及用数值（浮点数）表示经过的秒数。

在采用浮点数的情况下，让人担心的是浮点数可能无法表示所有的时刻信息。struct timeval 的大小是 64 位，虽然浮点数的大小也是 64 位，但是可以实际用于表示数值的只有尾数部分的 52 位。微秒部分最大也不到 100 万，用 20 位就可以表示。因此，到秒数部分能够用 32 位表示的 2038 年为止，浮点数尚且可以应付。

我比较重视代码的简洁程度，所以这次选择使用浮点数来表示时刻之间相减的结果。

另外还需要注意，除了时刻和时刻之间以外，时刻和数值之间也可以进行减法运算。字符串转换函数 string()能够根据允许省略的时区参数，用任意时区表示时刻。

用任意时区表示时刻

用任意时区表示时刻的处理需要费一些功夫才能实现，所以这里介绍一下。目前还不知道用什

么方法把某个时刻转换为任意时区的时刻（struct tm）。这是因为 UNIX 的时间函数只能对 UTC 和本地时间的其中一个进行操作。

这个处理看起来很难，其实稍微想想办法就可以实现。函数 get_tm 可以从 time_t 和 utc_offset 得到该时区的 struct tm，其实现如图 5-32 所示。

```
static void
get_tm(time_t t, int utc_offset, struct tm* tm)
{
  t += utc_offset;
  gmtime_r(&t, tm);
}
```

图 5-32　获取任意时区的 tm 的函数

实现竟如此简单，可能会让人感到有些扫兴。针对与任意时区的 UTC 只有时差部分不同的时刻，使用 gmtime_r(3) 就可以得到该时区的 struct tm 了。仔细想想的确如此，不过我还是感到有些意外。

时刻字面量

这样就在 Streem 中引入了时刻类型，我想趁这个机会再引入时刻字面量（直接记载数值和字符串的常量）。虽然拥有时刻字面量的编程语言并不多，但考虑到在数据处理中时刻处理的重要性，设置字面量也是一件很正常的事情。

这就涉及如何表示时刻字面量的问题了，可能的话我想使用 ISO 8601。不过遗憾的是，下面这种写法看起来像是整数的减法运算，所以不能直接使用。

```
2016-05-01
```

于是我参考了 JIS X 0301 的做法，在日期之间使用“.”来区分。也就是说，Streem 的时刻表示会变成下面这样。

```
2016.05.01
2016.05.01T00:00:00Z
2016.05.01T00:00:00.342Z
2016.05.01T00:00:00+09:00
```

虽然我觉得这么做很方便，但是时刻表示中包含了表示时区的加号和减号，这就可能会带来一些混乱。Ruby 中没有时刻字面量，它是通过下面这种形式的方法调用来生成时刻对象的。

```
Time.new(2016,5,1,0,0,0)
```

我不确定这种形式是否可行，因此感到很烦恼。不过既然好不容易实现了时刻字面量，就先予以保留，先使用一段时间看看，如果用起来感觉不好，以后也可以删除。这也是还没有用户的语言可以不用顾虑的地方。

时刻字面量的实现

时刻字面量的实现并不难。在语法分析器 lex.l 里添加解释时刻字面量的正则表达式之后，再添加表示时刻字面量的节点即可。

lex.l 实际修改的地方如图 5-33 所示。

```
DATE    [0-9]+\.[0-9]+\.[0-9]+
TIME    [0-9]+":"[0-9]+(":"[0-9]+)?(\.[0-9]+)?
TZONE   "Z"|[+-][0-9]+(":"[0-9]+)?
%%

{DATE}("T"{TIME}{TZONE}?)? {
  lval->nd = node_time_new(yytext, yyleng);
  LEX_RETURN(lit_time);
};
```

图 5-33　lex.l 修改的地方

之后使用 strptime(3) 等把字符串形式转换为 struct timeval 即可。灵活使用已实现的函数就不会有任何困难了。

CSV 的时刻支持

比时刻字面量更重要的是 CSV 的时刻支持。在预想的 Streem 用例中，日期和时刻数据最重要的来源就是 CSV 文件。

现在 Streem 可以自动判断 CSV 文件的各个字段是字符串还是数值，不过我还会在此基础上增加对时刻数据的支持。如果字段的值是 ISO 8601 形式或者 Streem 的时刻字面量形式，就把它当成时刻数据转化为时刻对象。

我以为实现很简单，可实际动手去做时，却发现浮点数的实现代码里有重大 bug，修复这个 bug 反倒费了不少事。

到这里才发现在这一系列开发中都没有被发现的重大 bug，看来是时候对语言的语法进行测试了。

小结

这次为 Streem 添加了支持时刻数据的功能。作为一名长年使用 UNIX 的程序员，我一直都对 UNIX 提供的功能和 API 感到很满意，不过这次我注意到它的时间相关的 API 还不够完整。尤其是这次要处理的既不是 UTC 也不是本地时间，而是时区，这就更加困难了。

时间和历法的处理本来就相当复杂，在哪里进行 API 的设计都很麻烦。Ruby 也不例外，Ruby 处理时刻的 `Time` 类和处理日期的 `Date` 类在设计时也考虑到了各种场景，因而变得相当复杂。

API 设计的相关内容可以参考田中哲的《API 设计案例学习》[①] 一书。这本书花了整整一章来介绍时间设计上的困难，了解这种设计过程的人读起来肯定会感同身受，不了解的人也可以抱着探索未知世界的心情去读一读这本书。

时光机专栏

完美地支持了多个时区

本节是 2016 年 6 月刊中刊登的内容。我们在思考时间是什么的时候，会认为它是一种从过去流向未来的非常简单的东西。此外，时间也经常作为一个基本的物理值出现，所以我们也容易把它看作一个数学意义上的值。

可是一旦在编程中处理时刻和时间，就会发现时间中竟然包含了许多与文化和政治相关的因素。比如，时刻的表示方法就与文化息息相关。各个国家和地区的时区和时差是由政治决定的，何时插入闰秒也是根据观测发现的差异通过会议来决定的。

UNIX 和对其进行标准化的 POSIX 原本似乎不太关心时刻和时间的处理，所以 API 不够丰富。这次我只使用 POSIX 的函数对处理多个时区的难题发起了正面挑战，虽然这个过程非常艰辛，但是比我预想中更加顺利，对此我感到心满意足。

① 原书名为『API デザインケーススタディ』，目前暂无中文版。——译者注

5-5 统计基础的基础

大数据是现在的一个非常热门的话题，其实数学在很早之前就致力于处理大型数据，这项工作被称为统计。本节将介绍如何在Streem中实现统计的基础部分。

让我有些难以启齿的是，我当学生的时候就不擅长数学。通常人们认为编程属于理工科，理工科的人应该很擅长数学，因此很多人对我不擅长数学感到意外，或者觉得我虽然不擅长数学，但是也不至于太差。

可实际上我是真的不擅长数学。上高中时，数学 III 的成绩得过“1”（当时采用的是 10 分制评分标准），高三第一学期的定期考试[①]的平均分只有 16 分。无论是高考的时候，还是考上大学之后，数学都让我非常痛苦。

现在回想起来，应该是我对数学和算术的手动计算（我觉得交给计算机做就好了）提不起劲来，才导致我对数学不感兴趣，让我经历了那么多挫折。我现在也依然认为不应该让容易出错的人类去做需要保证正确性的计算。

当然，计算机科学是以数学为根基的，与数学有着不可分割的关系，但这并不是说编程的所有内容都需要用到数学。很多编程活动都以把握用户需求为主，与数学并没有很大的关系。尤其是编程语言的设计和用户接口（UI）这些我很早就开始感兴趣的领域，更是很少用到数学。

我曾经说过“数学没什么用”这样的大话，但是在 Ruby 的开发过程中却屡屡遇到需要数学发挥重要作用的情况。在社区成员的帮助下（请他们指出错误），我也一点点地解决了不少问题，不过现在还是觉得自己不擅长数学。

刚才说了不少题外话，现在我们回到正题。实现了流编程的 Streem 最适合应用的场景恐怕就是数据处理了，比如读取 CSV 格式的测试成绩，并进行成绩处理。这样我就不能以自己不擅长数学为借口了，下面我们就来一起看一下数据统计处理的基础。

总和与平均数

首先来看一下小学数学水平的平均数。我问过上小学的女儿，她说平均数是小学五年级学的。我从女儿那里借来了教科书，书里是这样定义平均数的：

---

① 一般指每学期的期中和期末考试，根据学校的不同也会有差别。——译者注

```
平均数 = 总和 ÷ 个数
```

平均数表示一组数据中所有数据之和除以这组数据的个数。假如期中考试是 14 分，期末考试是 18 分，那么按照下面的计算过程，平均分就是 16 分。

```
(14+18)÷2=16
```

到 5-4 节为止，Streem 已经提供了计算流中数据个数的 `count()` 和计算流中数据之和的 `sum()`，把这些函数组合起来，计算平均数就变得简单了。`average()` 函数的实现如图 5-34 所示。

```
struct avg_data {
  double sum;
  strm_int num;
};

static int
iter_avg(strm_stream* strm, strm_value data) {
  struct avg_data* d = strm->data;
  d->sum += strm_value_flt(data);
  d->num++;
  return STRM_OK;
}

static int
avg_finish(strm_stream* strm, strm_value data) {
  struct avg_data* d = strm->data;

  strm_emit(strm, strm_flt_value(d->sum/d->num), NULL);
  return STRM_OK;
}

static int
exec_avg(strm_stream* strm, int argc, strm_value* args, strm_value* ret, int
avg) {
  struct avg_data* d;

  if (argc != 0) {
    strm_raise(strm, "wrong number of arguments");
    return STRM_NG;
  }
```

```
  d = malloc(sizeof(struct avg_data));
  if (!d) return STRM_NG;
  d->sum = 0;
  d->num = 0;
  *ret = strm_stream_value(strm_stream_new(strm_filter, iter_avg, avg_finish,
(void*)d));
  return strm_ok;
}
```

图 5-34 average() 的实现

用 exec_avg 函数创建任务，对每个元素执行 iter_avg 函数来计算总和，最后使用 avg_finish 函数让总和除以个数，以此求得平均数。处理被分割为组成流的各个任务，这一点可能难以理解，但其实所做的不过是进行平均数的计算而已。

总和计算中的陷阱

平均数的计算很简单吧。这是小学五年级的水平，当然简单。可是现实世界却很残酷，看起来这么简单的处理却隐藏着陷阱。

如上所述，平均数的计算方法是用总和除以个数，可是这个总和的计算方法里有一个陷阱，那就是误差。

由于计算机无法表示正确的实数，所以就用浮点数作为近似值来进行计算，误差便随之而来。而且浮点数中有两个与误差有关的陷阱。

其中一个陷阱是方便人类计算的值对计算机来说却不见得方便。比如 0.1 是一个非常简单的值，表示 1 除以 10 的结果，用浮点数使用的二进制是除不尽的。也就是说，必须计算到某一位停止，这样就产生了误差。

另一个陷阱是浮点数之间反复计算容易积累误差。如果只是计算几个数值的总和也许不会有什么问题，但假如是计算几万个或者几千万个数值的总和，所产生的误差可能就无法忽略了。比如图 5-35 是计算 1000 万个同一个数的平均数的程序。repeat() 函数用于创建按照第 2 个参数指定的个数生成第 1 个参数指定的值的流。

```
repeat(0.15,10000000) | average() | stdout # 运行结果
# 0.1499999999834609
```

图 5-35 产生误差的平均数计算

因为是计算同一个数的平均数，所以结果也应该是这同一个数，可实际上误差的累积会使数值产生微小的差异。

大家也许觉得总和的计算不过是单纯的加法运算而已，但是如果考虑到误差，就会发现实际上非常麻烦。

Kahan 算法

当然，计算机科学中有用于解决这种问题的方法。

Kahan 算法作为减小计算浮点数的总和误差的算法而为人所知。该算法通过把实施加法运算后丢失的后面几位数字转入下一次运算来补偿误差。图 5-36 是维基百科里的伪代码。

```
function kahanSum(input)
  var sum = 0.0
  var c = 0.0          ◀— 补偿用的变量，值为处理过程中丢失的后面几位数字
  for i = 1 to input.length do
    y = input[i] - c   ◀— 如果没有问题，则c为0
    t = sum + y        ◀— 如果sum很大，y很小，则y的后面几位数字就会丢失
    c = (t - sum) - y  ◀— (t-sum)相当于y的前面几位数字，所以减去y就可以得到y的后面几
                           位数字（符号是相反的）
    sum = t            ◀— 数学上c应该永远是0。注意积极进行优化的编译器!
  next i               ◀— 在下一次循环中考虑y丢失的后面几位数字
  return sum
```

图 5-36　Kahan 算法

出自 https://en.wikipedia.org/wiki/Kahan_summation_algorithm

使用这个算法对图 5-34 的实现进行修改，如图 5-37 所示，需要修改的只有 `struct avg_data` 和 `iter_avg` 函数。

```
struct avg_data {
  double sum;
  double c;
  strm_int num;
};

static int
iter_avg(strm_stream* strm, strm_value data) {
  struct avg_data* d = strm->data;
  double y = strm_value_flt(data) - d->c;
  double t = d->sum + y;
  d->c = (t - d->sum) - y;
```

```
  d->sum = t;
  d->num++;
  return STRM_OK;
}
```

图 5-37 average() 的改进
使用了 Kahan 算法

这样做虽然增加了计算量，但却可以抑制误差的产生。改进之后再执行图 5-35 的程序，无论重复多少次，都能得到正确的（与原数值相同的）结果。

不只是这个计算，其他包含浮点数的计算都会产生误差。在选择算法时，要尽量选择考虑了误差的算法。

平均数和方差（标准差）

通过平均数我们可以知道多个值的整体趋势，但平均数只是“平均”了整体的值，必定会丢失一些信息。假如每个班级有 20 人，A、B 两个班级参加满分为 100 分的考试，A 班所有人都是 50 分，B 班 10 个人 100 分，10 个人 0 分，那么两个班的平均分都是 50 分。

A 班和 B 班的成绩趋势完全不同，但从平均数上来看却没有区别。为了检测出这种不同，在平均数之外还需要把握数值的离散程度。数值的离散程度用标准差表示。假设 $x_1, x_2, \cdots, x_i$ 的平均数是 μ，标准差就是图 5-38 的公式所定义的方差的正平方根 σ。

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2$$

图 5-38 方差（标准差的平方）的定义

这个公式对（像我这种）不太擅长数学的人来说很难，用语言描述就是对各个值与平均数的差求平方，然后求和，最后除以个数。

用这个公式计算刚才两个班的成绩，得到的结果是：A 班所有人得分相同，标准差是 0；B 班一半人满分，一半人零分，标准差约为 51.3。从这里就可以看出平均数同样是 50 分，但性质完全不同。

流算法

根据图 5-38 的定义，要想求得标准差，就得先求出平均数，然后再计算各个值与平均数的差。如果什么都不考虑直接去实现，就需要在计算平均数时读取所有的值，之后计算标准差时再从头读一遍同样的值。比如图 5-39 就是一个循环两次去计算标准差的程序（出自奥村晴彦的《C 语言最新算法宝典》[①] 第 254 页）。

① 原书名为『C 言語による最新アルゴリズム事典』，目前暂无中文版。——译者注

```
int i, n;
float x, s1, s2;
static float a[NMAX];

s1 = s2 = n = 0;
while (scanf("%f" &x) == 1) {    /* 第1次循环 */
  if (n >= NMAX) return EXIT_FAILURE;
  a[n++] = x; s1 += x;
}
s1 /= n;                         /* 平均数 */
for (i=0; i<n; i++) {            /* 第2次循环 */
  x = a[i] - s1; s2 += x * x;
}
s2 = sqrt(s2/(n-1));             /* 标准差 */
printf("个数: %d 平均数: %d 标准差: %g", n, s1, s2);
```

图 5-39　标准差的计算

但是多次读取同样的数据实在是太浪费资源了，更何况在流处理的情况下，还需要把处理过程中的数据（可能会非常大）保存到某个地方。

为了避免造成资源浪费，可以使用流算法，这是一种只需逐个读取数据就能够进行处理的算法。我查了一下，发现流算法也可以用于标准差计算中。

根据《C 语言最新算法宝典》，用如图 5-40 所示的程序进行计算，只需从头读一遍数据，就能以较小的误差计算出标准差。

```
int i, n;
float x, s1, s2;

s1 = s2 = n = 0;
while (scanf("%f" &x) == 1) {
  n++;                           /* 个数 */
  x -= s1;                       /* 与计算过程中的平均数的差 */
  s1 += x / n;                   /* 平均数 */
  s2 += (n-1) * x * x / n;       /* 平方和 */
}
s2 = sqrt(s2/(n-1));             /* 标准差 */
printf("个数: %d 平均数: %d 标准差: %g", n, s1, s2);
```

图 5-40　标准差的流算法

我向 Streem 添加了使用这个算法计算方差和标准差的函数（`stdev()` 和 `variance()` 函数）。由于版面有限，这里就不贴出它们的代码了，这些代码的结构与图 5-34 的 `average()` 函数类似。请大家自行阅读 stat.c 源代码中 `exec_stdev()` 函数的相关部分。

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

用 Streem 计算标准差

接下来就使用新定义的 stdev() 函数来计算前面例题中的标准差。首先思考一下如何生成全班同学都得了 50 分的 A 班成绩。

由于全班 20 个人的成绩都是 50 分，所以只需连续生成 20 个值为 50 的数据即可。因此，这里需要使用前面介绍过的 repeat()。下面的式子就可以向标准输出写入 20 个 50。

```
repeat(50,20)|stdout
```

计算平均数的代码如下所示。

```
repeat(50,20)|average()|stdout
```

计算标准差的代码如下所示。

```
repeat(50,20)|stdev()|stdout
```

B 班得 100 分的有 10 人，得 0 分的也有 10 人，所以我们可以把两个流结合在一起。为了获得在 10 个 100 的后面跟着 10 个 0 的数据流，我们可以像图 5-41 那样编写程序，组合使用 repeat() 和 concat()。

```
concat(repeat(100,10),repeat(0,10))|stdev()|stdout
# 51.29891760425771
```

图 5-41　B 班成绩的生成与标准差的计算

但是现实生活中是不会出现这种全班同学分数都一样，或者一半满分一半 0 分的情况的。根据我们的经验，像考试分数这种非人造数据的分布规律通常是平均数附近的值最多，随着值与平均数的差距越来越大，值也会变得越来越少。

偏差值

虽然平均数和标准差可以用流算法进行计算，但是有些指标的计算却无法使用流算法。

比如排名和偏差值这两个统计成绩的指标就不能用流算法计算。计算排名时需要按照成绩进行排序，计算偏差值时需要先计算平均数和标准差。

作为示例，下面我们来计算一下偏差值。偏差值的定义如下所示。

```
偏差值 = （个人成绩 - 平均成绩）×10/ 标准差 + 50
```

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

用这个公式计算偏差值的代码如图 5-42 所示。

```
input = fread("result.csv")|map{x->number(x)}
avs = input | average()  # 平均成绩
sts = input | stdev()    # 标准差
zip(avs, sts) | each{ x ->
  avg = x(0); std = x(1)
  fread("result.csv") | map{x->number(x)}| each { score ->
    ss = (score-avg)*10 / std + 50
    print("个人成绩: ", score, "偏差值:", ss)
  }
}
```

图 5-42　计算偏差值

zip 那部分可能不太好理解，我来解释一下。从同一个输入流计算出了平均成绩和标准差。average() 和 stdev() 都返回只有一个元素的流。对两个流使用 zip() 函数，就可以取出元素合并为数组，所以接下来的 each() 不用于循环，而是用于对取出的值进行处理。

在读取一遍数据并计算平均数和标准差之后，如果还需要从头再读取一遍数据，就太让人反感了。就算无法避免重复读取数据，也应该会有更好的指定方法吧。我稍微思考了一下，觉得使用 future 和 promise 说不定可以改进这个问题。我把这个当成作业以后再去研究。

排序

根据值的大小对数据进行排序是计算机科学的一个重要课题，为了提高排序速度，人们想出了各种算法，其中快速排序（quick sort）被认为是目前最快的算法。

C 标准库里提供了 qsort(3) 函数，可以简单地对内存中的数据进行排序。不过这个函数有一个严重的问题，就是只能对不超过内存大小的数据进行排序。

这个问题暂且可以用外部归并排序（merge sort）的方法解决。外部归并排序是指将数据分割为内存可以容纳的大小后再进行读取，然后对各个数据进行排序并写入文件里，之后从前面开始读取每个已排好序的文件，把数据连接起来完成整体的排序，具体步骤如下所示。

1. 从原来的数据集读入大小不超过内存的数据
2. 对已读入的数据进行排序，并写入文件
3. 重复第 1 步和第 2 步，直到完成所有的数据处理
4. 从已写入的多个文件中分别读取 1 个元素
5. 将最小的元素写入结果文件
6. 从已写入的元素的来源文件中再读取 1 个元素，重复该处理，直到把数据全部写入结果文件

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

UNIX 的 sort 命令就是使用这个算法实现了排序。

很难进行大规模的排序

但是外部归并排序也存在一些问题。第一个问题是创建工作文件会消耗硬盘容量。数据规模越大，消耗的硬盘容量也就越大。使用外部归并排序就是为了处理很大的数据，这样一来，容量的消耗就越发让人担心。除了注意配置合适的磁盘之外，可能也没有什么解决办法了[①]。

另一个问题是，Streem 中任何数据都可以在流中流动，成为排序的对象。也就是说，如果只是数值和字符串，那么就可以轻易地写入工作文件，但是有结构的数据就很难在不丢失信息的情况下写入文件了。使用 JSON 和 MessagePack 这种能够表示有结构的数据的格式可以在一定程度上解决这个问题。

可以看出，数据规模越大，需要考虑的事情也就越多。

说了这么多，我们还是早点进入实现阶段吧。这次先实现内存中的排序。虽然可能会让大家失望，但是关于使用外部归并排序来支持大规模数据排序的实现，我打算以后再去研究。

这里使用下面的代码对数据进行排序。从 input 把数据一次性读取到内存中，然后进行排序，之后把结果一次性写入 output。

```
input | sort() | output
```

如果需要排序的数据全部是数值，那么自然可以进行排序，不过也有各个数据都是数组，需要根据数组的第 n 个元素进行排序这样的情况，这时就需要指定函数。

也就是说，为了能够根据数组的第 1 个元素（由于索引从 0 开始，其实是第 2 个元素）排序，需要指定如下所示的比较函数。

```
sort{x,y->cmp(x(1),y(1))}
```

排序的应用

实现排序之后，我们就可以取出对统计来说有意义的值了。最容易理解的就是排名，排名就相当于根据成绩进行排序。

另外，排序后就可以得到中位数了。中位数是指一组有序数据中处于正中间的值。由于数据个数为偶数时没有正中间的值，所以就需要取最中间的两个值的平均数作为中位数。求中位数时使用 median()。

① 不过英文版的维基百科中提到，通过 in-place 的方式进行外部归并排序，可以把所需的磁盘容量限制在与原有数据同等大小的程度，但是这个说明却被维基百科标记为“缺少来源”。

中位数有点像平均数，但它在不受极端值的影响这一点上优于平均数。平均数受所有值的影响，如果由于测量错误而出现了极端值，就可能使误差变大。而中位数则几乎没有这样的问题。

抽样

虽然“大数据”这个词现在很流行，但是实际上规模过大的数据处理起来很困难，而且数据的处理开销也很高，有时甚至很难收集到足够多的数据。

而统计这门学问原本就是为了对难以收集实际数据的“大数据”进行推算而诞生的。

当总体（population）的数量达到一定程度时，只需要用很少的样本（sample）就可以把握整体的趋势。如果允许有 5% 左右的误差，那么在总体有 10 万人的情况下，把握整体趋势所需要的样本人数只有 383 人。

为了更容易处理数据，Streem 中也实现了用于抽样的函数。用于抽样的流算法有蓄水池抽样（reservoir sampling）。

蓄水池抽样按照如下步骤进行抽样。要得到 N 个样本，需要：

1. 把开头的 N 个样本添加到数组中
2. 对之后的第 i 个元素，生成 0 到 $i-1$ 之间的随机数 r
3. 如果随机数 r 比 N 小，则将数组中的第 r 个元素替换为第 i 个元素

用开头的 N 个样本填满蓄水池，之后每当上游有元素流过来时，就以 counter/size 的概率随机替换部分元素。通过替换蓄水池中的元素，最终可以实现从总体中取出样本的效果。

有些读者可能会觉得代码比文字说明更容易理解，因此我准备了用 Ruby 编写的蓄水池抽样算法的代码（图 5-43）。

```
def reservoir_sampling(seq, k)
  e = seq.to_enum
  reservoir = e.take(k)
  n = k
  e.each do |item|
    r = rand(n)
    n += 1
    if r < k
      reservoir[r] = item
    end
  end
  return reservoir
end
# 调用
print reservoir_sampling(0..1000000, 10)
```

图 5-43 用 Ruby 编写的蓄水池抽样算法

我使用这个算法实现了用于抽样的 `sample()` 函数。例如在管道中添加 `sample(100)` 语句，就可以从整个流中随机取出 100 个元素，并发送给下游处理。即便总体特别庞大，也能够在保持整体趋势的情况下选出元素。

前面提到过，如果允许有 5% 左右的误差，即使总体庞大，也可以根据少数样本来把握数据的整体趋势。不过需要注意的是，如果在不了解总体大小的情况下对数据进行筛选，就可能无法得到正确的结果。

小结

在大数据备受瞩目的当今社会，统计变得越来越重要。真希望未来 Streem 也能像 Excel 那样成长为容易操作的统计分析工具。

时光机专栏

自己不擅长的领域真想交给别人去做

本节是 2016 年 7 月刊中刊登的内容。在这一节中，我们引入了几个统计函数。

正文中也提到过，我对数学总有一种畏惧的心理，所以写这一节时费了很多心力，我甚至向读小学的女儿借了教科书，一边复习一边痛苦地进行写作。

不管是并发编程还是数学处理，都不是我想自己动手去完成的，所以我非常希望用完成度很高的工具来实现这些处理。遗憾的是，我想要的工具都不存在，所以只好自己去开发。既然自己去做，就难免需要直面一些自己不想去碰的问题，内心真的很抗拒。如果是跟擅长这些问题的人组队开发可能会好一些，可这样的人偏偏是不容易遇到的。可能是因为我不善于与人沟通吧。

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

5-6 随机数

通过扔骰子获得的随机数经常在游戏中使用，而在Streem进行的数据处理中也常常用到这样的随机数。本节我们就来学习随机数的实现和应用的基础知识。

随机数没有规则，我们事先不知道会得到哪个数。比如扔骰子会得到 1 到 6 之间的整数，但我们并不知道具体会得到哪一个。据说为了让骰子的每个面朝上的概率均等，骰子上的点的深浅都经过了细心的调整。所以当扔骰子的次数达到一定程度时，各个面朝上的概率是相等的。听说便宜的骰子在制作时没下这么多功夫，各个面朝上的概率会不同。

在计算机中也有许多场景需要用到随机数，比如大部分游戏就以某种形式用到了随机数。除了游戏之外，ssh 和 https 的加密通信等也会用到随机数。

蒙特卡罗法是在数据处理领域使用随机数的一个典型例子。蒙特卡罗法是使用随机数进行计算的方法，例如用随机数求圆周率（图 5-44）。

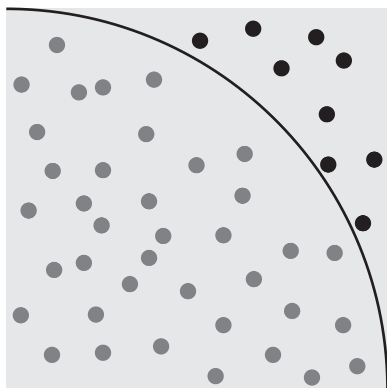


图 5-44　蒙特卡罗法的示例

在正方形中放置大量的由随机数确定坐标的点，这些点就会被分为 1/4 圆之内的点和圆外的点。用包含在圆内的点的个数除以点的总数，得到的值就约等于 π/4，而且点越多值越精确。这就是用蒙特卡罗法求圆周率的方法。

除了蒙特卡罗法以外，5-5 节介绍的蓄水池抽样也使用了随机数。

另外还有使用随机数提高效率的随机算法（randomized algorithm）。随机算法中有一个通过允许误差的存在来提高效率的布隆过滤器（bloom filter）。

真随机数和伪随机数

其实计算机无法像扔骰子那样得到随机数（真正的随机数）。但即便不是真正的随机数，计算机也可以通过某种计算得到“近似于随机数的数”。

最简单的方法就是使用时间信息。也就是说，使用当前的秒值，或者使用以微秒、纳秒为单位的时刻信息来得到随机的数值。据说当年的 Z80 微机就是把保存刷新内存的时间信息的 R 寄存器的值（1 ~ 127）当成随机数使用的。

我们也可以通过计算得到伪随机数。下面是几种具有代表性的伪随机数生成算法。

- 线性同余法
- 梅森旋转法
- Xorshift

通过计算求得的伪随机数具备重现性，这也是它的一个特征。也就是说，用同样的初始值开始计算就能得到完全相同的随机数序列。

重现性与随机数的“没有规则，事先不知道会得到哪个数”的特性相矛盾。不过伪随机数的重现性有时也会为我们带来便利。比如在使用随机数模拟某种场景的情况下，我们可以使用伪随机数通过同样的初始值得到同样的随机数序列，这就意味着能够重现完全相同的模拟结果。在对模拟结果进行试验等情况下，这个重现性就会起到非常重要的作用。

伪随机数的评估

前面提到过生成伪随机数的算法有很多种，那么这些算法都有什么区别呢？我们要如何评估这些算法呢？

伪随机数加密算法的评估标准有以下几点。

- 偏差
- 周期
- 速度（计算量）
- 密码学上的安全性

偏差表示算法生成的伪随机数与真正的随机数之间的差异程度。一些算法生成的随机数会与真正的随机数之间存在偏差，比如生成的随机数是某个特定数值的倍数等。

通过计算求得的伪随机数序列很容易陷入重复同一个模式的局面。在这种情况下，我们把从开始到相同模式再次出现为止的这一段长度称为周期。周期短的伪随机数算法的缺点是容易预测下一个值，而且偏差过大。

偏差和周期是伪随机数加密算法固有的特点。至于为什么每个算法都有这些特点，这是由数学决定的，对其原因进行讲解就超出我的能力范围了。大家知道这个特点就可以了。

速度表示计算下一个随机数所需要的计算量。统计和模拟等领域需要用到大量的随机数，在这种情况下，如果伪随机数算法的计算量过大，那么如何缩短计算时间就成为了整个处理的难题。

密码学上的安全性表示该算法是否可以安全地用于密码学领域。在密码学领域，密钥的生成以及一次性密钥的实现等许多场景都需要用到随机数。如果随便用了不可靠的伪随机数，就会出现漏洞，密码就有可能被破解。

为了实现密码学上的安全性，生成的随机数序列不仅要没有偏差，还需要确保密码不会被破解。虽然这些条件很难满足，但也存在 Blum-Blum-Shub 这种在密码学上安全的伪随机数算法。另外，把加密时使用的单向函数用在通过普通的伪随机数算法生成的随机数上，就能实现密码学上的安全性。但不管怎么说，确保安全性的这些方法都会产生一定的开销，所以用在普通场景（比如统计等）中会有些小题大做。

本节介绍的伪随机数算法，如果不加修改直接使用，在密码学上都不是安全的。这里大家只要记住在加密的场景中使用随机数时，不可以随便使用一般的伪随机数算法即可。

线性同余法

下面我将对（密码学上不安全的）伪随机数算法中具有代表性的算法依次进行说明。

首先要介绍的就是线性同余法。据说线性同余法是被广泛使用的伪随机数算法之中最古老的算法。图 5-45 的递推表达式定义了使用线性同余法产生的随机数序列。

$$Xn+1=(A\times Xn+B)\bmod M$$

图 5-45　线性同余法的递推表达式

在这个表达式中，A、B、M 是常量，$M>A$、$M>B$、$A>0$、$B\geqslant 0$。这些常量和初始值 X_0 的选取方法将决定随机数序列的性质。

线性同余法的周期最大也不过是 M。不过这个特点会受常量的选取方法的影响，如果选了不合适的常量，周期就可能会比 M 短得多，偏差也可能变大。

我为像我一样不擅长数学的读者准备了用 C 语言编写的使用线性同余法产生随机数的程序（图 5-46）。这个程序选定的常量如下所示，这是被公认为比较好的一种常量的组合方式。

```
A = 1566083941
B = 1
M = 2^32
```

```
uint32_t
rand(void)
{
  static uint32_t seed = 1;
  seed = seed * 1566083941UL + 1;
  return seed;
}
```

图 5-46　用 C 语言编写的通过线性同余法产生随机数的程序

虽说线性同余法不是计算量特别大的算法，但它产生的随机数质量并不高。尤其需要注意的是后面几位数字的随机性较低。比如用线性同余法得到了 32 位随机数，如果要从这个数中获取 0 ~ 7 的随机数，就需要用 `r` 表示 32 位随机数，用 `r >> 29` 取出前面几位数字作为随机数，而不能使用 `r%8` 或者 `r&0xf` 这样的操作获取后面 4 位。

另外，线性同余法周期比较短，而且从递推表达式可以明显地看出，在得到某个随机数之后，下一个随机数的值也就确定了。因此，假如在蒙特卡罗法中使用线性同余法，点可能就会分布不

均，呈现出格子状。游戏自不用说，在统计和模拟等领域使用这个算法时也需要注意。现在有几种伪随机数算法要比线性同余法更好用，希望大家可以去了解一下。

C 的标准库里提供了获得随机数的 `rand()` 函数，不过在大多数情况下使用的是线性同余法（C 的 ISO 标准没有规定 `rand()` 函数在随机数计算上要使用线性同余法，只是标准里的参考代码使用的是线性同余法）。

也就是说，在直接使用 `rand()` 函数时，很有可能碰到前面所说的线性同余法的问题。在某些情况下，我们也许就不能使用系统提供的 `rand()` 函数，而是需要自己编写伪随机数生成函数了。

梅森旋转法

与线性同余法相比，梅森旋转法这一伪随机数加密算法的出现时间较晚，该算法是在 1997 年由当时的广岛大学的松本真和西村拓士开发的。

梅森旋转法最大的特征是周期长。在讲解线性同余法时也提到过，如果周期较短，在模拟等需要使用大量随机数的场景下随机数就会发生偏差。梅森旋转法的周期是 $2^{19\,937}-1$，非常长。这个数即使用十进制表示也超过了 6000 位，是一个非常大的数。$2^{19\,937}-1$ 属于梅森素数，这也是这个算法名称的由来。

这个算法不仅周期长，而且连续的随机数之间的相关关系也较小（专业的说法是“多维均匀分布”），比以前的伪随机数生成算法的速度更快，非常好用。

这些优点实际上也得到了人们的认可，包含 Ruby 和 Python 在内的很多编程语言都采用了这个算法作为标准的随机数生成算法。

梅森旋转法虽然很好用，但也不是没有缺点，它的其中一个缺点就是保存内部状态的向量太大。线性同余法只用了 1 个整数来保存内部状态，而梅森旋转法却足足用了 623 个 32 位整数来保存。如果要保存处理过程中的状态并重现随机数序列，使用梅森旋转法就比较麻烦了。而且最重要的是，在对这 623 个状态向量进行初始化时，一不小心随机数的质量就会变低。

SFMT（SIMD-oriented Fast Mersenne Twister）算法改进了这些缺点。该算法号称速度是原始的梅森旋转法的 2 倍，不过可能是因为该算法在 2006 年才出现，还比较新（虽然也有 10 多年了），没怎么见到有人使用。我自己也是为这次写稿做调研的时候才知道的，以后有机会可以尝试使用一下。

Xorshift

Xorshift 是乔治・马萨格利亚（George Marsaglia）在 2003 年发表的，比梅森旋转法出现的时间更晚。

Xorshift，顾名思义，是一个只使用 xor 运算（逻辑异或）和 shift 运算（位移）就能快速得到伪随机数的算法，因此可以快速地计算随机数。

虽然 Xorshift 的周期比梅森旋转法短（根据状态向量大小的不同，周期在 $2^{32}-1$ 到 $2^{128}-1$ 之间

变化），但是在随机性上比线性同余法好得多。不仅如此，它的实现也非常简单。Xorshift 最简单的实现（状态向量为 64 位）如图 5-47 所示，图 5-47 代码产生的随机数的周期是 $2^{64}-1$。

```
uint32_t xorshift(void) {
  static uint64_t x = 88172645463325252ULL;
  x = x ^ (x << 13); x = x ^ (x >> 7);
  return x = x ^ (x << 17);
}
```

图 5-47　Xorshift 的实现

没想到通过这么简单的计算就可以生成质量这么高的随机数，而且这个算法居然到了 21 世纪才被人发明出来，真是令人意外。

Xorshift 还派生出了随机性更高的算法，即 Xorshift* 和 Xorshift+。

伪随机数的初始值

正如前面介绍的那样，伪随机数算法通过计算来产生（看起来）随机性高的随机数序列，但这只不过是根据计算结果产生的伪随机数，并不是真正的随机数。

计算机基本上是按照指示工作的，也就是说，从同样的状态开始就会得到同样的结果，所以很难产生真正的随机数。

通过尽量把难以预测的值作为伪随机数算法的初始值使用，就可以产生更像随机数的伪随机数。

非常典型的初始值是时刻。使用从操作系统得到的当前时刻的精确的部分（以微秒或纳秒为单位），就可以在每次运行时都得到随机的初始值。

不过需要注意的是，硬件实际搭载的时刻功能，并不能像操作系统一样以微秒和纳秒为单位对时刻进行分解。即便系统调用可以返回以微秒为单位的时刻，返回的时刻也不一定是正确的以微秒为单位的值。

有的操作系统也会提供一些其他的方法来得到更加随机的数值。

/dev/random

从外部获取的用户输入等，基本上是不可预测的，因此这些输入具备作为随机数基础的“熵”（混乱性）。

比如 Linux 系统从驱动等收集基于外部信息的熵，然后通过读取名为 /dev/random 的设备文件，消费收集到的熵，得到“真正的随机数”。

不过从设备等收集到的熵是有限的，所以从 /dev/random 中读取过多的随机数会耗尽收集到的熵。在这种情况下，/dev/random 就会阻塞读取，一直保持等待的状态，直到收集到熵。

原本只是想得到随机数而已，没想到却产生了阻塞，这可能会给我们带来一定的困扰。在这种情况下，我们可以使用另外一个名叫 /dev/urandom 的设备文件，这个文件是不会产生阻塞的。/dev/urandom 在耗尽熵时，会以过去的熵为初始值，使用密码学上安全的伪随机数算法返回随机数。

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

其他操作系统也提供了类似的功能，比如 FreeBSD 也提供了 /dev/random。这些操作系统返回随机数的功能相同，只不过 FreeBSD 一开始使用的就是密码学上安全的伪随机数算法，不会产生阻塞。从这一层面来说，FreeBSD 的 /dev/random 就相当于 Linux 的 /dev/urandom。

从 /dev/random 得到的随机数要比时刻更难预测，很适合作为初始值使用。实际上 Ruby 就使用了 /dev/urandom（如果可用的话）来初始化随机数序列。

而 mruby 和 Streem 在初始化随机数时使用的是时刻信息。由于 /dev/urandom 只能在 Linux 等部分操作系统上使用，考虑到可移植性，所以使用了时刻信息。不过对于还没怎么考虑可移植性的 Streem 来说，采用 /dev/urandom 可能会更好。

也许有人会认为需要产生随机数时干脆全部从设备文件中读取就好了，但是从性能方面考虑，这种做法不能替代伪随机数算法。

伪随机数的基准测试

程序员在选择使用某种方法时，往往会实际写出代码比较一下，下面我们就来测试一下本节介绍的各个伪随机数算法。如果实现无误的话，偏差和周期等理论上是固定的，所以这次就先对性能进行评测。

首先，使用线性同余法、梅森旋转法和 Xorshift 这 3 种算法生成 1 亿个随机数，比较它们的运行时间。

实际的基准测试代码如图 5-48 所示，图 5-49 是输出结果。表 5-12 是评测时使用的机器的配置。

```
#include <stdio.h>
#include <inttypes.h>
#include <sys/time.h>

/* linear congruential method */
uint32_t
lcm_rand(void)
{
  static uint32_t seed = 1;
  seed = seed * 1566083941UL + 1;
  return seed;
}

/* merseene twister */
#define N 624
#define M 397
#define M 397
#define MATRIX_A 0x9908b0dfUL   /* constant vector a */
```

```
#define UPPER_MASK 0x80000000UL /* most significant w-r bits */
#define LOWER_MASK 0x7fffffffUL /* least significant r bits */

static uint32_t mt[N]; /* the array for the state vector  */
static int mti=N+1; /* mti==N+1 means mt[N] is not initialized */

void
mtw_init(uint32_t s)
{
  mt[0]= s & 0xffffffffUL;
  for (mti=1; mti<N; mti++) {
    mt[mti] = (1812433253UL*(mt[mti-1]^(mt[mti-1]>>30))+mti);
    mt[mti] &= 0xffffffffUL;
  }
}

uint32_t
mtw_rand(void)
{
  uint32_t y;
  static const uint32_t mag01[2]={0x0UL, MATRIX_A};

  if (mti >= N) { /* generate N words at one time */
    int kk;

    if (mti == N+1)   /* if mtw_init() has not been called, */
      mtw_init(5489UL); /* a default initial seed is used */

    for (kk=0;kk<N-M;kk++) {
      y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
      mt[kk] = mt[kk+M] ^ (y >> 1) ^ mag01[y & 0x1UL];
    }
    for (;kk<N-1;kk++) {
      y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
      mt[kk] = mt[kk+(M-N)] ^ (y >> 1) ^ mag01[y & 0x1UL];
    }
    y = (mt[N-1]&UPPER_MASK)|(mt[0]&LOWER_MASK);
    mt[N-1] = mt[M-1] ^ (y >> 1) ^ mag01[y & 0x1UL];
    mti = 0;
  }

  y = mt[mti++];
```

```
  /* Tempering */
  y ^= (y >> 11);
  y ^= (y << 7) & 0x9d2c5680UL;
  y ^= (y << 15) & 0xefc60000UL;
  y ^= (y >> 18);

  return y;
}

uint32_t
xor_rand(void) {
  static uint64_t x = 88172645463325252ULL;
  x = x ^ (x << 13); x = x ^ (x >> 7);
  return x = x ^ (x << 17);
}

#define TIMES 100000000
#define BENCH(name) do {\
  struct timeval tv, tv2, tv3;\
  int i;\
  char *f;\
  name ## _rand();                   /* rehearsal */\
  f = #name;\
  gettimeofday(&tv, NULL);\
  for (i=0; i<TIMES; i++) {\
    name ## _rand();\
  }\
  gettimeofday(&tv2, NULL);\
  timersub(&tv2, &tv, &tv3);\
  printf("func %s: %ld.%06ldsec\n", f, tv3.tv_sec, tv3.tv_usec);\
 } while (0)

int
main()
{
  printf("benchmark repeats %d times\n", TIMES);
  BENCH(lcm);
  BENCH(mtw);
  BENCH(xor);
}
```

图 5-48 随机数生成的基准测试程序

从基准测试的结果来看，速度最快的是线性同余法，生成 1 亿个随机数耗费了约 0.3 秒，其次是 Xorshift，耗费了约 0.7 秒，耗费时间最长的是梅森旋转法，约 0.96 秒。

```
benchmark repeats 100000000 times
func lcm: 0.305532sec ←— 线性同余法
func mtw: 0.963983sec ←— 梅森旋转法
func xor: 0.733444sec ←— Xorshift
```

图 5-49 随机数生成的基准测试的结果

线性同余法的随机数质量不高，我们先将其排除在外。综合考虑周期的长短和性能，也许根据场景区分使用梅森旋转数和 Xorshift 比较好，或者也可以考虑使用号称速度是梅森旋转法的 2 倍的 SFMT。

表 5-12 基准测试机器的配置

信息	值
机型	Thinkpad E450
CPU	Core i7-5500U
CLOCK	2.40 GHz
MEM	16 GB
OS	Ubuntu 16.04
GCC	gcc 5.4.0

其实我也想过对 SFMT 进行基准测试，只是它的代码里用到了 `SSE2` 等 `SIMD` 命令，源代码要比想象中复杂。由于时间所限，这次没能将 SFMT 以简洁的形式加入到基准测试程序里，这让我感到非常遗憾。

Xorshift 并没有达到相当于梅森旋转法的 2 倍的速度，所以如果 SFMT 真的像它宣称的那么快的话，就成了兼顾周期和性能的最强算法。现在我还没有完全理解这个算法，尚不能熟练使用，今后我会继续研究它的。

Streem 的随机数功能

在当前的 Streem 中，与随机数相关的功能有生成随机数流的 `rand()` 函数，以及使用随机数进行抽样的 `sample()` 函数。

`rand()` 函数是逐个传递随机数的流。比如下面的程序在运行后，就会持续输出随机数，直到输入中断让它停止为止。

```
rand()|stdout
```

`sample()` 函数会按照参数指定的数量从上游流中取出元素进行抽样。

```
fread("data.csv")|sample(100)|stdout
```

上面这行代码会从 `data.csv` 包含的行中取出 100 行进行抽样，并显示在标准输出中。即使 `data.csv` 有几万行，`sample()` 函数也会均等地进行抽样，这就是蓄水池抽样算法的魔力。

这些函数采用了伪随机数算法 Xorshift 的修正版 Xorshift64*（周期是 $2^{64}-1$）。Xorshift64* 与原版 Xorshift 不同的是在随机数生成的最后阶段使用了乘法。尽管这样做会让计算速度降低，但是随

机性得到了加强，而且 Xorshift64* 也通过了伪随机数算法测试 Die Hard 的所有测试项目。

只是现在的 Xorshift64* 周期太短，还不适合在实际的统计处理等场景中使用，今后可能会将其替换为梅森旋转法或 SFMT。

随机数的各种类型

之前介绍的随机数都是均匀随机数，即某个范围内的数会以均等的概率出现，但是统计和模拟所需要的随机数并不都是均匀随机数。

用于统计分析的 R 语言中内置了多种随机数生成函数（表 5-13）。命令规则是“`r+` 分布的名称（多为省略形式）”。比如均匀随机数是服从均匀分布（uniform distribution）的随机数，所以是 `runif`，服从正态分布（normal distribution）的随机数是 `rnorm`。

表 5-13　R 的随机数生成函数

名称	含义	解释
`runif`	均匀随机数	也就是随机数
`rnorm`	标准正态随机数	正态分布随机数
`rbinom`	二项随机数	二项分布随机数
`rpois`	泊松随机数	泊松分布随机数
`rexp`	指数随机数	指数分布随机数
`rgamma`	伽马随机数	伽马分布随机数

Streem 的 `rand()` 相当于 `runif`。至于其他的随机数生成函数，我打算在需要时再去实现，不过标准正态随机数因为用法明确，而且可以立刻在模拟等场景中使用，所以我想早点去实现。不过函数的名字很难取，我纠结于是根据正态分布取名为 `nrand()`，还是取名为 `rand_normal()`。虽然前者比较简洁，但是考虑到今后要提供各种各样的随机数生成函数，还是不要省略过多为好，因此我最后使用了 `rand_norm()`。

小结

本节讲解了生成随机数的伪随机数算法及其应用，特别是在以数据处理为主的 Streem 中，随机数起到了重要的作用。在读者看到本篇之前，我打算继续实现 Streem 的随机数功能。

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

时光机专栏

我挑战了新的算法

本节是 2016 年 10 月刊中刊登的内容，主要介绍了随机数生成的相关内容，这是继 5-5 节的统计基础之后又一个偏数学的话题，让我很痛苦。我尽自己所能讲解了各种算法及其评估方法。

传统的随机数生成算法中有线性同余法，比较新的算法里比较有名的有梅森旋转法。不过我想尝试新的东西，所以这次采用了 Xorshift 算法。Xorshift 比较简单，而且速度很快，生成的随机数质量也很高，但是它有很多派生出来的算法，再加上网络信息错综复杂，让我很伤脑筋。我数学不好，自己无法鉴别哪些信息是正确的，因此不知道该相信哪个才好，这让我非常为难。

我姑且以自己认为是正确的信息进行了实现，但这真的是正确的吗？其实我自己也没有把握。

5-7 数据流图

本节将介绍适合流处理语言Streem的从流的输入到图形输出的过程。GUI库淘汰的速度很快，所以我们以能够长久使用的CUI为基础来输出图形。本节也将介绍改善外观的方法。

相信很多工程师都和我一样，喜欢什么都测一测。我每天晚上都会称体重，生病发烧时也是每隔 15 分钟就测一次体温，以把握体温的变化趋势。我的计算机屏幕上也会显示内存使用量、CPU 使用率和网络流量等图表。《宇宙战舰大和号》的舱内有相当多的计量表，估计也是出于相似的心理吧。

所以我考虑让 Streem 也能把输入的数据图形化。可是图形的处理非常麻烦，GUI（Graphical User Interface，图形用户界面）的 API 因平台而异，如果要使用某个 GUI 库，光是讲解这个库就得占掉整个版面。

而且一般来说，GUI 库的寿命要比编程语言短得多。看看 Ruby 早期非常受欢迎的 GUI 库 `Tk` 的命运就知道了，这个库现在已经基本上不见踪影了。

所以我打算先不讨论 GUI 这种和美观相关的部分，只讲解如何显示图形这一本质话题。

GUI 和 CUI（或者叫 CLI）

CUI 是相对于 GUI 的一个专业用语，是 Character User Interface（字符用户界面）的简称。不过在日本以外的国家基本听不到 CUI 这种说法，它似乎已经成了日本独有的一个术语，其他国家常用的是 CLI（Command Line Interface，命令行界面）。

CUI 使用字符来显示界面，能够在已经使用了几十年的终端中工作，今后也不用担心终端会消失。这次我就使用 CUI 来开发图形显示的功能。

stag

于是我开始寻找可供参考的工具，然后就发现了 stag。

stag 这个工具用于从标准输入读取数值数据，然后输出为条形图。比如图 5-50 这段代码可以把 Streem 生成的随机数序列输出为图表，输出结果如图 5-51 所示。

```
streem -e 'rand_norm()|take(100)|stdout' | stag
```

图 5-50　使用 stag 生成随机数图表

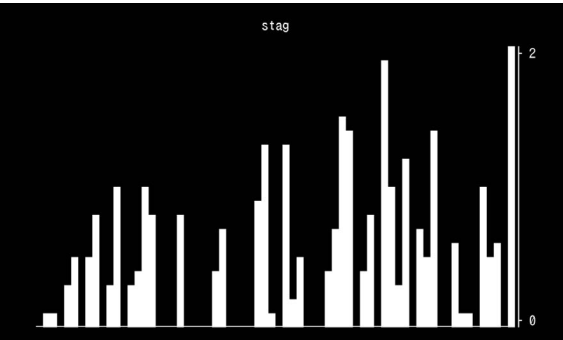


图 5-51　使用 stag 生成随机数图表的输出结果

stag 虽然是一个很方便的工具，但我觉得还是不要从 Streem 发送数据到其他进程，而是直接生成图表会更加方便，所以本节我会向 Streem 添加与 stag 作用相同的函数。

画面构成

首先把 stag 显示的画面按照图 5-52 进行分割。

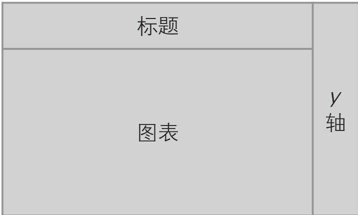


图 5-52　stag 的画面分割

通过字符把这些部分输出即可。幸运的是大部分终端支持使用转义字符串对输出的文字设置颜色，或者移动光标。使用这个功能，还可以替换部分画面。

stag 使用了用于 CUI 的 `ncurses` 库，但是本次开发用不到太多的功能，所以我决定直接使用转义字符串来画图。

转义字符串

最近听到“终端”这个词，只是想到“输入 shell 命令的窗口”。可是很久以前说起“终端”（设备），那就是指用于计算机输入输出的机器。现在我们使用的“终端”只是用软件实现了过去终端机器的功能，准确地说应该称为“终端模拟器”。

在终端作为机器使用的那个年代，大型计算机（现在看来其实性能非常弱）上通常会连接几个终端进行操作。这是个人计算机出现之前的事情。

那个年代的终端只有显示文本的显示屏和输入信息的键盘，没有计算机的处理能力。它的功能仅限于把输入的信息全部发送到计算机（当时称为主机），然后在画面上显示返回的信息。

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

只是这样也太过简陋，后来人们通过向终端发送转义字符加上一些字符的方式，使终端能够调用操作画面的各种功能了。这样的字符串被称为转义字符串，比如下面的字符串可以清空画面。

```
ESC [ 2 J
```

这些转义字符串根据设备的厂商和机型的不同而有所不同，但当时普及度很高的 DEC 的 VT100 机型的转义字符串基本上成为了标准。

这次在图表功能的实现中使用的转义字符串如表 5-14 所示。

表 5-14　具有代表性的转义字符串

转义字符串	含义
ESC [x;y H	将光标移动到 (x, y)
ESC [2 J	画面清空
ESC [1 K	清空光标前面一行
ESC [3x m	指定文字颜色（黑红绿黄青紫蓝白）
ESC [0 m	重置颜色
ESC [6 n	获取光标位置
ESC [? 25 l	隐藏光标
ESC [? 25 h	显示光标

获取窗口大小

首先获取当前的终端窗口的大小。获取窗口大小的方法有很多，这次我们使用 ioctl。

ioctl 是以文件描述符为对象控制输入输出的系统调用。

调用形式如下所示，内核根据 request 进行控制。

```
ioctl(fd, request, ...);
```

即使对同一个 request，处理也可能会因文件描述符所代表的设备而异，不过我们不用在意具体的设备，可以把它当成某种面向对象。UNIX 的优点是可以通过文件描述符把各种对象按照面向对象的方式进行处理，这个操作系统刚出现时给人带来了一种耳目一新的感觉。

使用 ioctl 向内核询问窗口大小的请求是 TIOCSWINSZ，参数是保存窗口大小的结构体 struct winsize 的指针（图 5-53）。

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <termios.h>

static int
get_winsize(int* row, int* col)
{
  struct winsize w;
  int n;
```

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

```
  n = ioctl(1, TIOCGWINSZ, &w);
  if (n < 0 || w.ws_col == 0) {
    return -1;
  }
  *row = w.ws_row;
  *col = w.ws_col;
  return 0;
}

int
main()
{
  int row, col;
  int n;

  n = get_winsize(&row, &col);
  if (n < 0 || col == 0) {
    printf("WINSZ failed\n");
    exit(1);
  }
  printf("WINSZ (%d, %d)\n", w.ws_col, w.ws_row);
}
```

图 5-53　使用 ioctl 获取窗口大小

移动光标

我们使用转义字符串来移动光标。如果输出的文字描述符指向的是终端，那么发送转义字符串就可以使光标自由移动。发送下面的转义字符串可以把光标移动到坐标 (x, y) 的位置。移动的原点是左上角，需要注意的是原点的位置不是 (0,0)，而是 (1,1)。

```
ESC [ x;y H
```

图 5-54 的程序表示在清空画面后，移动光标，输出 Hello World，运行程序后的输出结果如图 5-55 所示。

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

static void
clear()
{
  printf("\x1b[2J");
}

static void
move_cursor(int row, int col)
{
  printf("\x1b[%d;%dH", row, col);
}

int
main()
{
  int i;

  clear();
  for (i=1; i<10; i++) {
    move_cursor(i, i*2);
    printf("%d:Hello World\n", i);
  }
}
```

图 5-54 移动光标的示例

```
1:Hello World
  2:Hello World
    3:Hello World
      4:Hello World
        5:Hello World
          6:Hello World
            7:Hello World
              8:Hello World
                9:Hello World
```

图 5-55 移动光标的示例的输出结果

如果既可以获取画面大小，又能清空画面，还能移动光标，那么之后把这些功能组合起来就可以输出图形了。

绘制标题

标题的绘制很简单。先指定标题，然后移动到画面第一行，绘制标题。要想让标题显示在画面中间，就需要考虑画面大小和字符串长度来移动光标。

图 5-56 是显示标题的程序（的 `main` 部分）。图 5-56 的程序无法单个编译，因为它使用了图 5-53 和图 5-54 的程序中定义的函数，通过这个程序应该可以理解图 5-53 和图 5-54 的函数的用法。

```
int
main(int argc, char **argv)
{
  int i, row, col;
  char* title;
  int tlen;
  int start;

  if (argc != 2) exit(1);
  title = argv[1];
  tlen = strlen(title);
  // 获取画面大小
  if (getwinsize(&row, &col) < 0) exit(1);
  start = (col - tlen) / 2;
  clear();                    // 清空画面
  move_cursor(start, 1);      // 移动光标
  write(1, title, tlen);      // 显示标题
  move_corsor(row-1, 1);      // 移动到最后一行
  return 0;
}
```

图 5-56　显示标题

绘制图表

图表的绘制也比较简单。根据图表数据、最大值和窗口大小，从每行的开头开始一个字一个字地判断该位置（列）是否应该显示图表，如果需要显示图表，就为该位置设置颜色。每行的右边显示 y 坐标。

graph_bar() 函数

图 5-57 是集上述内容之大成的 `graph_bar()` 函数，其中的 `get_winsize()`、`move_cursor()` 和 `clear()` 等函数与图 5-53 和图 5-54 中定义的相同。

```
struct bar_data {
  const char *title;
  strm_int tlen;
  strm_int col, row;
```

```
  strm_int dlen, llen;
  strm_int offset;
  strm_int max;
  double* data;
};

static void
show_title(struct bar_data* d)
{
  int start;

  clear();
  if (d->tlen == 0) return;
  start = (d->col - d->tlen) / 2;
  move_cursor(1, start);
  fwrite(d->title, d->tlen, 1, stdout);
}

static void
show_yaxis(struct bar_data* d)
{
  move_cursor(1,2);
  printf("\x1b[0m");    /* 恢复颜色 */
  for (int i=0; i<d->llen; i++) {
    move_cursor(i+3, d->dlen+1);
    if (i == 0) {
      printf("├ %d   ", d->max);
    }
    else if (i == d->llen-1) {
      printf("├ 0");
    }
    else {
      printf("│");
    }
  }
}

static void
show_bar(struct bar_data* d, int i, int n) {
  double f = d->data[i] / d->max * d->llen;

  for (int line=0; line<d->llen; line++) {
```

```
    move_cursor(d->llen+2-line, n);
    if (line < f) {
      printf("\x1b[7m ");  /* 颜色反转 */
    }
    else if (line == 0) {
      printf("\x1b[0m_");  /* 恢复颜色，绘制基线 */
    }
    else {
      printf("\x1b[0m ");  /* 恢复颜色，绘制空白 */
    }
  }
}

static void
show_graph(struct bar_data* d)
{
  int n = 1;

  show_yaxis(d);
  for (int i=d->offset; i<d->dlen; i++) {
    show_bar(d, i, n++);
  }
  for (int i=0; i<d->offset; i++) {
    show_bar(d, i, n++);
  }
}

static int
init_bar(struct bar_data* d)
{
  if (getwinsize(&d->row, &d->col))
    return STRM_NG;
  d->max = 1;
  d->offset = 0;
  d->dlen = d->col-6;
  d->llen = d->row-5;
  d->data = malloc((d->dlen)*sizeof(double));
  for (int i=0;i<d->dlen;i++) {
    d->data[i] = 0;
  }
  show_title(d);
```

```
  return STRM_OK;
}

static int
iter_bar(strm_stream* strm, strm_value data) {
  struct bar_data* d = strm->data;
  double f, max = 1.0;

  if (!strm_number_p(data)) {
    strm_raise(strm, "invalid data");
    return STRM_NG;
  }

  f = strm_value_float(data);
  if (f < 0) f = 0;
  d->data[d->offset++] = f;
  max = 1.0;
  for (int i=0; i<d->dlen; i++) {
    f = d->data[i];
    if (f > max) max = f;
  }
  d->max = max;
  if (d->offset == d->dlen) {
    d->offset = 0;
  }
  show_graph(d);
  return STRM_OK;
}

static int
fin_bar(strm_stream* strm, strm_value data) {
  struct bar_data* d = strm->data;

  move_cursor(d->row-2, 1);
  if (d->title) free((void*)d->title);
  free(d->data);
  free(d);
  return STRM_OK;
}

static int
```

```
exec_bar(strm_stream* strm, int argc, strm_value* args,
 strm_value* ret) {
  struct bar_data* d;
  char* title = NULL;
  strm_int tlen = 0;

  strm_get_args(strm, argc, args, "|s", &title, &tlen);
  d = malloc(sizeof(struct bar_data));
  if (!d) return STRM_NG;
  d->title = malloc(tlen);
  memcpy((void*)d->title, title, tlen);
  d->tlen = tlen;
  if (init_bar(d) == STRM_NG) return STRM_NG;
  *ret = strm_stream_value(strm_stream_new(strm_consume
r, iter_bar, fin_bar, (void*)d));
  return STRM_OK;
}
```

图 5-57　graph_bar() 函数的实现

下面是各部分的处理步骤。首先，在初始化部分进行以下处理。

- 获取画面大小
- 清空画面
- 绘制标题

每当收到来自上游流的数值数据时，就进行以下处理。

- 保存数据
- 计算最大值
- 绘制图表

绘制图表时的操作如下所示。

- 绘制 y 轴
- 从左边开始显示条形图

只要正确使用转义字符串，这些操作就都不是很难。

这里让我花了点心思的是（用于显示图表的）缓冲的使用方法。`bar_data` 结构体中有一个名为 `data` 的数组，该数组保存了与能够显示的图表的数量相应的数据，`offset` 用于在指定的位置写入输入的数据。`offset` 在超过数组长度 `dlen` 时会恢复为 0，所以这是一种环形缓冲（ring buffer）。

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

显示时先显示从 offset 开始到缓冲结束为止的数据，然后再显示开头到 offset 之前的数据，这样就可以按照输入的时间顺序从左开始显示图表了。

绘制图表时，移动光标，在值没有到达某一高度时用反转色填充空白，剩余部分用普通颜色填充。

绘制图表的代码写好之后，我们来画几个图表看看效果。以下代码的显示结果如图 5-58 所示。

```
seq(100)|graph_bar()
```

以下代码的显示结果如图 5-59 所示。

```
rand_norm()|take(100)|graph_bar()
```



图 5-58 seq(100)|graph_bar() 的显示结果

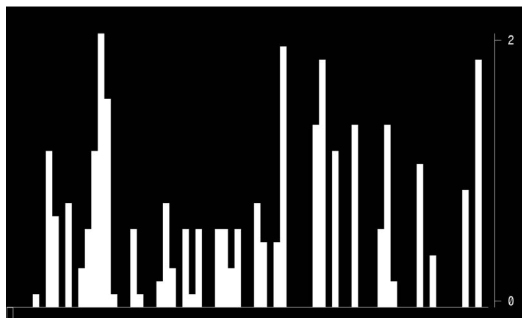


图 5-59 rand_norm()|take(100)|graph_bar() 的显示结果

调整窗口大小

以前的机器终端的画面大小是不会发生变化的，而现在的终端模拟器的画面大小则会根据窗口大小的调整而发生变化。

CUI 也支持窗口大小的调整。这次因为要让源代码精简一些，所以就不让 Streem 支持窗口大小的调整了，不过我会在这里为大家讲一下具体的做法。

在终端大小发生变化时，系统会向终端进程发送 SIGWINCH 信号。收到信号时，重新初始化图表，再次绘图即可。

接收信号时使用 singal 函数。下面的代码表示在接收到信号时，处理函数指定的中断处理（interrupt handler）会被调用。

```
singal(信号编号, 处理函数)
```

实际的调用例子如下所示。

```
singal(SIGWINCH, winch_handler);
```

由于无法预料中断会在何时被发送，所以有的中断处理中的逻辑无法执行。因此，通常的做法是，在中断处理中只对变量进行赋值，由主程序检查是否有中断。

不过要把这个函数引入到 Streem 里还存在一些问题。在一个进程中只能对一个信号注册一个处理函数，所以如果要在程序的其他地方使用信号，就会导致某个处理不会被调用。

我们也可以换个角度思考。反正开销也不大，干脆每次都重新获取窗口大小好了，如果与之前的大小不一样，就重新进行初始化。这样一来，我们就可以不依赖信号来解决问题，而且语言处理器的其他部分也无须在意信号处理。

光标的事后处理

说起信号，还有一个让人担心的问题。为了防止绘制图表时光标闪烁，现在的 `graph_bar` 的做法是在绘制过程中隐藏光标。

但问题是在通过“Ctrl+C”组合键触发键盘中断而中止程序时，光标还是处于消失的状态。要解决这个问题，就需要对键盘中断发生时所发送的 SIGINT 信号的处理进行设置。

但正如在介绍 SIGWINCH 时所说的那样，在信号处理的设定中，不能在程序的其他部分对同一个信号指定另外的处理。一般来说，通过对整体进行协调而开发的应用软件中基本上不会有问题，但在像语言处理器那样各个功能独立的情况下，就会出现问题了。

真是让人头疼。

但是仔细想一想，问题的根源在于对一个信号只能设置一个处理，所以我在 Streem 处理器中准备了设置信号处理的函数。

```
strm_signal(sig, func, arg)
```

上述新函数会对 `sig` 指定的信号设置处理函数 `func`。即使指定多个处理函数也不会发生覆盖，所有处理函数都会被调用。`arg` 是 `void*` 类型的参数，会被传递给处理函数。

使用这个函数，就不必担心在 Streem 处理器的其他地方处理相同的信号时发生冲突了。

今后的课题

到这里就可以从数值流输出条形图了。不过用 21 世纪的眼光来看，用字符来显示图形实在是太落伍了，所以自然要去思考有没有什么更好的办法，以使图形更加美观。

Sixel 图像库

不过基于前面提到的原因，我还是想尽量避免使用 GUI 库，Sixel 图像库这项技术正符合我的需求。Sixel 是“Six Pixels”的缩写，这项技术把终端上的一个字符分解为 6 个像素，并通过转义字符串为每个像素指定 256 种颜色。虽然 Sixel 在 20 世纪 80 年代就已经被 DEC 的 VT200 系列所使用，是一项比较古老的技术，但是在解释它的终端模拟器上，则可以通过扩展功能显示 1600 万色（最多）。另外，Sixel 虽然是使用了转义字符串的效率不太高的图形显示方法，但是在现在的机器上也可以流畅地显示 GIF 动画。

Sixel 图像库的缺点是没有得到所有终端的支持，比如我手头用的 xterm 和 mlterm 支持 Sixel 图像库，但 gnome-terminal 和 roxterm 却不支持。

小结

本节实现了将数值流作为输入，通过字符图形输出条形图的函数。老实说图形还很粗糙，期待今后能够加以完善，比如使用 Sixel 以像素为单位进行显示等。

时光机专栏

今后也会继续开发Streem

本节是 2016 年 11 月刊中刊登的内容，实现了以字符为基础的图形显示功能。在处理完数据之后，我们经常会想用图形来显示，但我一直觉得把数据导入 Excel，再使用它的图形功能进行显示这种做法很麻烦，本节实现的功能就是为和我一样的人准备的。

不过，光是实现一个粗糙的条形图就已经让我拼尽全力了。虽然还不能说已经完成了图形功能，但是我想这至少证明了还是有希望实现这个功能的。

到目前为止，我已经多次向 Streem 添加功能，这里就先告一段落。Streem 离完成还很遥远，所以我不会就此停止开发，只是连载不能无限期地继续下去了。本书也就到此告一段落，Streem 今后会以 OSS 的形式继续开发下去。

后记

正文里提到过，我不擅长数学，而且对并发编程也不拿手，但是在这个大数据和数据科学盛行、计算机逐渐多核化的时代，这些我不擅长的领域的重要性却在与日俱增。

我们中的多数是工程师。工程师的灵魂不就是用技术去解决问题吗？那么如何解决这些问题呢？我给出的答案是使用 Streem。挑战自己不擅长的领域是极其困难的，即使有灵光闪现，也很难做出成绩。我在本书中也几度表达了自己的懊悔——由于实力不足而遗留下几个没能彻底解决的问题，以及因为干劲和时间都不足而没能改善完成度很低的代码，但我认为这些“缺点”不会降低 Streem 语言本身的价值。

其实我非常喜欢 Streem 的运行模型，在 2015 年的 Ruby 大会和 RubyConf 大会上，我都提议在 Ruby 将来的版本（Ruby 3）里引入以 Streem 的并发模型为基础的功能。虽然我比较积极，但是经过讨论，最终可能会采用笹田耕一提议的 Guild 模型。选择这个模型的原因是它在表现能力和兼容性上显示了强大的实力。

Streem 的并发模型的抽象度很高，无法以极细的粒度编写处理代码，这既是它的优点，也是它的缺点。不过如果要把这个并发模型引入 Ruby 3，我们就不能对它的缺点视而不见。

我考虑的引入 Streem 模型的方式是让一种语言里存在两种模型，也就是让非常相似的两种语言混合在一起，以此来统一 Ruby 和 Streem 各自不同的运行模型，不过这样可能会给用户带来很大的混乱，不被采用也是没有办法的事情。

没有做到的事情

我在正文中多次表达了懊悔之情，这也说明 Streem 离完成还差得很远，特别是没有很好地实现 GC 功能是个大问题。现在的 Streem 处理器几乎不会释放内存，所以在处理大量数据时内存必定会消耗殆尽。

说起来，在 4-4 节我介绍过提高效率的方法，就是让每个线程在自己的空间内分配对象，GC 也以线程为单位进行，但在实际开发时却发现实现该方法要比想象中困难。在管道里交付数据时，数据对象会传给下一个线程。如果这个对象包含了对数组等其他对象的引用，就需要把被引用的对象递归地复制到下一个线程的内存空间。我一开始没想到会这么复杂，所以就推迟了这个实现。

但要想使 Streem 成为实用的语言处理器，就得想办法解决这个问题。第一步我准备修改使用了 NaN Boxing 技术的对象表示方法，然后引入 `libgc`。

`libgc` 是通过链接的方式向 C 语言编写的应用中添加 GC 功能的库，不过它有一个限制条件，

就是不能改变指针的值，而现在的 Streem 使用了 NaN Boxing，不受制于这个限制条件。

根据目前被广泛使用的浮点数运算标准 IEEE 754，在双精度浮点数的 64 位中，符号位使用 1 位，指数位使用 11 位，尾数位使用 52 位。

另外，IEEE 754 中还引入了作为未定义的计算结果（比如被 0 除）的 `NaN`。“指数部分全部是 1”的值被定义为 `NaN` 值。虽然表示 `NaN` 的值只有一个就够了，但是实际上指数部分全是 1 的值都是 `NaN`。如果好好利用这一点，用 `NaN` 解释的值中就有 52 位可以用来塞入整数值，这就是 NaN Boxing 技术。实际上，这 52 位中有 4 位被用于表示值的类型的标签，剩下的 48 位被用于保存实际的值。

如果只有 48 位可用，我们可能就会觉得在 64 位操作系统中无法保存指针，但实际上除了一些特殊情况以外，包括 Linux 在内的大多数操作系统即使在 64 位机器上也只用 48 位来表示指针值，所以这一点大家大可放心。

但是这种做法会把指针的值转换成浮点数，导致 `libgc` 无法发现指针，不能进行 GC，所以需要改进 NaN Boxing 的部分，修改格式使指针值依然是指针。

想法非常简单。在 64 位指针值中，实际用于表示地址的只有 48 位，剩余 16 位都是 0。NaN Boxing 在这个部分插入标签，将其伪装成 `NaN` 值。也就是说，需要表示指针值时，把标签的值调整为 0 就可以了。当然，调整后的值不是真正的浮点数，所以就需要在作为浮点数取出时对值进行加工，不过这并不会产生什么大的开销。

这种 NaN Boxing 的变体被称为 Favor Pointers。

把 NaN Boxing 改为 Favor Pointers 之后就简单了，只需要用 `GC_malloc()` 把现在调用 `malloc()` 分配内存的部分替换掉，就能够实现 GC 了。虽然 `libgc` 不能像 4-4 节介绍的那样利用应用相关的固有知识进行优化，但是它提供了支持线程的分代 GC，因此可以有效地进行 GC。

后会有期

虽然 Ruby 3 中不采用 Streem 的并发模型，但这并不意味着 Streem 就没有价值了。我想把 Streem 当成一个独立的语言继续开发下去。

其实还有一些与 Streem 理念相似的语言和工具，比如 tab 和 datamash。我会尽量让 Streem 成长为这些语言和工具的竞争对手。

还请大家期待我的下一部作品！

松本行弘

2016 年 12 月

版权声明

MATSUMOTO YUKIHIRO GENGO NO SHIKUMI

written by Yukihiro Matsumoto.

Copyright © 2016 by Yukihiro Matsumoto.

All rights reserved.

Originally published in Japan by Nikkei Business Publications, Inc.

Simplified Chinese translation rights arranged with Nikkei Business Publications, Inc.

through CREEK & RIVER Co., Ltd.

本书中文简体字版由 Nikkei Business Publications, Inc. 授权人民邮电出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究。



微信连接



回复“IT人文”查看相关书单



微博连接

关注@图灵教育 每日分享IT好书



QQ连接

图灵读者官方群I：218139230
图灵读者官方群II：164939616

图灵社区
iTuring.cn

在线出版，电子书，《码农》杂志，图灵访谈

为什么要创建一门新语言？

新语言开发中遇到了什么困难？如何解决？

C、Java、Python等语言的设计为什么是现在这样？

什么样的语法不会给使用者造成负担？

……


Ruby之父全面披露新语言开发的整个过程，详细解说语言设计的各种细节


从主要语言的常见功能到少数语言的独特功能，穿插对现有语言特征的介绍，干货满满


巧设“时光机专栏”，坦言不足之处及再思考，展现编程大师的“平凡”一面

图灵社区：iTuring.cn
热线：(010)51095183转600
分类建议 计算机/编程语言
人民邮电出版社网址：www.ptpress.com.cn



ISBN 978-7-115-51616-9
定价：89.00元

图灵社区会员 ChenyangGao(2339083510@qq.com) 专享 尊重版权

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这可以找到我们：

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks